GEOMETRIC NANOCONFINEMENT EFFECTS ON THE ELECTRONIC AND MECHANICAL PROPERTIES OF SELF-ASSEMBLED MOLECULAR SYSTEMS

A Dissertation

by

BRADLEY WILLIAM EWERS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	James D. Batteas
Committee Members,	Karen L. Wooley
	Steven E. Wheeler
	Andreas A. Polycarpou
Head of Department,	David H. Russell

December 2014

Major Subject: Chemistry

ABSTRACT

With the ongoing research and development of nanoscale technologies and materials, it becomes increasingly important to understand how local environment influences molecular and material properties. An important factor in this regard is geometric nanoconfinement, for example, the restriction of molecules to nanostructure surfaces. The bulk or average characteristics of materials and molecules do not appropriately define their behavior in these circumstances, and highly localized measurement techniques developed to specifically identify the influence of confinement on their properties is essential to understanding their characteristics and behavior.

In this dissertation, two forms of geometric confinement are considered in the context of different molecular properties. First, the role of radial confinement on the tribological properties of self-assembled monolayers (SAMs) is considered. SAMs are an excellent model lubricant for experimental studies of boundary lubrication, and they have been employed as boundary lubricant additives and surface coatings. The lubricated contacts of technologically relevant surfaces, however, consist of asperity interactions, and the summit curvature of these asperities can impact the critical cohesive forces from which the properties of the SAM are derived. Molecular dynamics simulation was employed to understand the influence of nanoscopic surface curvature, as well as surface coverage density, two factors which together contribute to the cohesive forces of SAMs, on their tribological properties. In particular their dissipative potential and effective surface protection were examined, as well as the influence of these factors on the contact

mechanics of functionalized nanoasperity contacts.

Another mode of geometric confinement studied in this work is two-dimensional nanoconfinement of molecules and its influence on the mechanism of charge transport in molecular systems. Effective control of charge transport in molecules is essential for molecular modification of CMOS technologies, and is critical in controlling charge carrier dynamics in dye-sensitized photovoltaics. In this work, the size dependence of the electronic properties of thiol-tethered zinc porphyrin aggregates on the Au(111) surface was investigated. AFM nanolithography was used to confine these molecules within an alkanethiol matrix on the Au(111) surface, forming molecular islands of specific dimensions to investigate the relationship between island size and charge transport, demonstrating a shift from tunneling based charge transport to the more tunable and efficient charge hopping based transport.

ACKNOWLEDGEMENTS

Many people have contributed to the completion, though none more than my graduate advisor Dr. James D. Batteas. His guidance, support, and friendship has been invaluable to my successful completion of this endeavor, and I thank him for allowing me and trusting me to follow my own course on the research projects on which I've engaged. I would also like to thank the members of the Batteas research group. In particular, I would like to thank Dr. Amanda E. Schuckman for serving as my mentor, helping me to develop the necessary knowledge and techniques, and for tolerating my occasional distraction and helping me stay focused in the early stages of my graduate studies. I would also like to thank Ms. Alison A. Pawlicki, for whom I played the role of mentor. Though she came in with virtually no understanding of chemistry, her dedication allowed her to overcome this challenge to become an effective researcher. I learned as much if not more about mentoring new students as she learned chemistry and technical skills from me, and these are skills I look forward to carrying with me as I pursue a career mentoring graduate students in chemical research. Finally, I'd like to thank Ms. Carrie B. Werke, who provided friendship and emotional support throughout my graduate school experience.

I would also like to thank those who made my path to graduate school possible. My father devoted his time and energy to my education, taking the time to read to me on his lap when I was a child and supporting my various forays into computer science, engineering, and science as I grew older. Having started a family of my own, I have begun to truly recognize the sacrifices he took on my behalf in terms of time, energy, and

financial support, and my path to graduate school and completion of my Doctoral thesis would have been much more challenging without his perseverance and support.

Finally, I would like to thank my wife for accepting this enormous sacrifice so early in our lives together. I don't think either of us knew what we were getting ourselves into, but she has handled the restrictions on my time, the stress, and the uncertainty with grace. She proved invaluable in helping me maintain the discipline and focus necessary to complete this work. Also, she brought home most of the bacon.

NOMENCLATURE

11-MUA	11-Mercaptoundecanoic Acid
16-MHA	16-Mercaptohexadecanoic Acid
3P1P	3-phenyl-1-propanol
AFM	Atomic Force Microscopy
AMBER	Assisted Model Building with Energy Refinement
BKS	van Beest, Kramer, van Santen Silica Potential
CCD	Charge-coupled Device
CHARMM	Chemistry at Harvard Molecular Mechanics
CHIK	Carré, Horbach, Ispas, Kob Silica Potential
CMOS	Complementary Metal-Oxide Semiconductor
CP-AFM	Conductive Probe Atomic Force Microscopy
CRLS	Contact Radius-to-Line Step Ratio
CVD	Chemical Vapor Deposition
DCM	Dichloromethane
DDT	Dodecanethiol
DFT	Density Functional Theory
DLC	Diamond-like Carbon
DMT	Derjaguin-Muler-Toporov Contact Mechanics
FTIR	Fourier Transform Infrared Spectroscopy
HOMO	Highest Occupied Molecular Orbital

IFM	Interfacial Force Microscopy
JKR	Johnson-Kendall-Roberts Contact Mechanics
LAMMPS	Large-Scale Atomic/Molecular Massively Parallel Simulator
LUMO	Lowest Unoccupied Molecular Orbital
MD	Molecular Dynamics
MEMS	Microelectromechanical Systems Device
NDR	Negative Differential Resistance
ODT	Octadecanethiol
OPE	oligo(phenylene ethynylene)
OPLS	Optimized Potentials for Liquid Simulations
OPV	oligo(p-phenylenevinylene)
OTS	Octadecyltrichlorosilane
PFD	Perfluorodecalin
RESPA	Reference System Propagator Algorithm
ROM	Range-of-Motion
SAM	Self-assembled Monolayer
SFA	Surface Force Apparatus
SFG	Sum frequency generation spectroscopy
SNOM	Scanning Near-field Optical Microscopy
SPM	Scanning Probe Microscopy
STM	Scanning Tunneling Microscopy
STS	Scanning Tunneling Spectroscopy

TGA	Thermogravimetric Analysis
THF	Tetrahydrofuran

TABLE OF CONTENTS

Page

Page

Page

LIST OF FIGURES

Page

Page

Page

Page

Page

LIST OF TABLES

Page

CHAPTER I

INTRODUCTION

1.1 Overview

The miniaturization of technology is broadly driven by the needs of society for faster, more efficient devices and device capabilities. For many decades, this pursuit has successfully led to the miniaturization of electronic and mechanical devices, delivering ever improving performance in an information based society that demands speed and reliability with an increasing need for energy efficiency, however only recently have we begun to approach the limits of device miniaturization. Barring enormous advances in the physical sciences, we are intrinsically limited to the dimensions of molecules and atoms, and as devices are miniaturized to these length scales, a detailed understanding of how local inhomogeneity alters the characteristics, behavior, and reliability of these systems becomes necessary. For example, Intel is, today, capable of mass producing transistors on silicon based microchips on the order of 22 nm in size, and projects the ability to create transistors on the order of 10 nm in size in the near future.[1] This size reduction features up to a 2-fold increase in the surface-area-to-volume ratio, implying 2-fold greater sensitivity of these devices to their local environment, requiring ever greater precision in device manufacturing to produce consistent results.

The theme of nanoconfinement stems from the notion that, as materials and devices become sufficiently small, it becomes impossible to consider them in terms of only their intrinsic bulk characteristics and, rather, material shape and dimension as well as the local

chemical environment, must also be considered as an inseparable component of the material properties. This has been considered extensively in the scaling down of solid state materials, with the size dependence of the electronic properties of metal nanoparticles and semiconductor quantum dots well-established. The use of molecules in nanoscale systems is, on the other hand, a largely different problem. A single molecule is already one of the smallest devices or machines currently conceivable, but the way we approach molecules is completely different from solid state materials. They are inherently nanoscale and strongly influenced by their environment, but molecules are generally characterized in terms of their average environment. When molecules are used in a nanoscale context, however, this treatment does not suffice, not only do outliers have an outsized influence, our knowledge of the molecular dynamics and energetics must be tailored to their specific environment and application.

In this work, the effects of geometric nanoconfinement on molecular systems are considered in two specific contexts and with two different properties in mind. First, the effects of radial confinement of SAMs on their lubricating capacity at sliding interfaces is considered. SAMs are an excellent model for boundary lubricants, species that protect surfaces during intermittent contact, but to understand their role as lubricants it is necessary to focus on their properties where surfaces come into contact, which, due to surface roughness, is at nanoasperities. Thus, while much is known about the general behavior and structure of SAMs on surfaces, in this application, it is necessary to understand their chemical, mechanical, and dissipative characteristics in a radially confined geometry. In order to reliably achieve this goal, it is necessary to understand the

nature of surface contact at technologically relevant interfaces, as well as how friction and wear evolve in lubricated contacts.

Second, the role of two dimensional nanoconfinement of a thiol-tethered zinc porphyrin molecule on its electronic properties is considered. Here, two dimensional nanoconfinement is used as a tool to explore how nearest-neighbor interactions can be made to influence the mechanism of charge transport through molecular systems. Charge transport is critical in a variety of systems, from biological energy harvesting systems to artificial photovoltaics, to the more exotic notion of replacing or enhancing semiconductor technology with devices based on organic molecules. Achieving this goal requires an understanding of the mechanisms of charge transport, and the conditions under which they occur.

1.2 Local Geometric and Chemical Effects on Molecular Film Lubricity

1.2.1 Friction and Wear at Interfaces

Friction and wear of contacting and sliding interfaces imposes enormous costs upon society. It has been estimated that these processes result in economic productivity losses on the order of 1.5-2% of global gross domestic product, or hundreds of billions of dollars.[2] It is therefore imperative that these processes are understood and the means to control them are developed and improved. Unfortunately, there are many barriers to effective research and development in this field, perhaps the most obvious being that these processes occur at solid-solid interfaces, which constitute the most challenging interfaces to probe experimentally. Additionally, a fundamental understanding of a macroscopic contact between surfaces requires characterization over length scales ranging from

Ångstroms upwards to centimeters and meters, resulting in enormous challenges in investigating and modelling these processes. Finally, the mechanical and chemical properties of sliding interfaces are highly sensitive to a variety of factors, including both the chemistries of the bulk materials and their surfaces, and how environment conditions and sliding of the surfaces alter the chemistry of the interface. Resolving all of these challenges is an ongoing process that combines both experimental and computational research, from the macroscale to the atomic scale.

The work described herein focuses on the atomic and molecular scale contributions to friction. To properly grasp how this fits in the broader issue of surface lubrication, it is necessary to develop an understanding of what exactly friction is, how it evolves in the contacts between surfaces, and under what conditions surface and lubricant chemistry affect the mechanical properties of the sliding contact. Furthermore, the attention of this work is focused on the smallest of contacts, so-called nanoasperity contacts, and to understand their importance it is necessary to understand how surfaces come into contact, over what length scales this occurs, and the magnitude of the surface forces and pressures experiences in these contacts.

1.2.2 Friction from the Macroscopic to the Nanoscopic Scale

Friction is perhaps one of the most elusive of the many forces we seek to understand. Unlike conservative forces in which particles or objects move in relation to a well-defined force field, friction is a dissipative force, a result of the second law of thermodynamics. Macroscopically, friction dissipation can simply be perceived as the result of surface deformation or heat generated at a sliding interface. At the atomic scale, it is more

appropriately viewed as disordering of atomic translational degrees of freedom, the conversion of uniform translational energy to disordered translational energy that can occur in the complete absence of permanent changes to the interface.[3] As the atoms of surfaces slide against one another, they experience a corrugated potential landscape representing a competition between interfacial interactions and bulk restoring forces, and the evolution of these forces result in strain and relaxation cycles that give rise to energy dissipation. The number of degrees of freedom that exist at the atomic scale render it virtually impossible to develop a well-defined, mechanistic understanding of these processes that is applicable to all systems, and much of what we know about friction is the result of observation, beginning with the fundamental principles of friction put forth by Amonton and Coulomb.[4] They observed that for macroscopic sliding objects, the force of friction is proportional only to the normal force at the sliding interface, regardless of the apparent contact area or sliding speed. This observation is straightforward and predictive, but without a detailed understanding of the underlying processes that give rise to friction, it is useless in rational design and optimization of materials, lubricants, and coatings designed to control the friction response of interfaces because it yields only two parameters, the material and environment dependent coefficient of friction, and the normal force. One might wonder why the friction coefficient of steel and rubber are so different, or what the mechanism of action of a lubricant is at a sliding interface, and these are not questions that can be resolved from this empirical explanation of the friction force.

A key result that has proven much more useful in understanding the friction response of materials is that, though friction is not a function of the *apparent* contact area, it is a

function of the *real* area of contact. Technologically relevant surfaces, i.e. those not formed in a laboratory with a specific surface morphology, are naturally rough over many length scales. When they come into contact, contact occurs only across a fraction of the apparent interface, and much of the material dependence of the friction coefficient of materials can be explained by this simple observation. Rubber, for example, is soft and deforms to accommodate the pressure at a contacting interface, providing a greater real contact area that increases rapidly with increases in applied load, resulting in a larger coefficient of friction than harder materials.

Connecting the macroscale and microscale views of friction has presented a considerable challenge. It has been proposed that for rough surfaces, which consists of a statistical ensemble of asperity contacts, the real contact area is linearly related to the applied load. Various models of rough surfaces have been developed which support this conclusion. Greenwood and Williamson[5] for example considered a rough surface as an ensemble of statistical asperities with statistical variation in asperity height, and Bush et al,[6] similarly considered a rough surface as an ensemble of parabaloids in which curvature and summit height were correlated. In both cases, reasonably linear relationships between contact load and real contact area could be achieved.

Though these models do suggest agreement, real surfaces are generally not found to have these prescribed morphologies. To address this discrepancy, Persson and coworkers have used the "self-affine fractal" model of surfaces,[7] a description which accommodates the roughness of surfaces at all length scales, and an example of a model self-affine surface topography is depicted in Figure 1.1. This peculiar definition is best understood

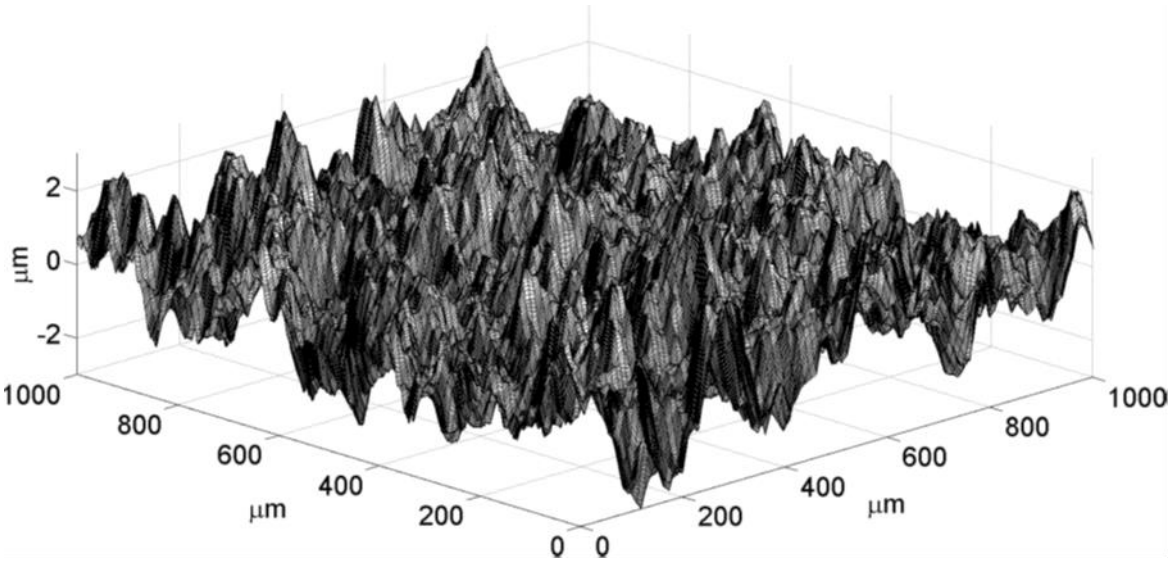


Figure 1.1. A model self-affine surface exhibiting roughness at all discernible length scales. Reprinted with permission from David, R.; Neumann, A. W. "Contact Angle Hysteresis on Randomly Rough Surfaces: A Computational Study". *Langmuir*. **2013**, 29, 4551-4558.[8] Copyright 2013 American Chemical Society.

in terms of the roughness power spectrum:[9]

$$C(q) = \frac{1}{(2\pi)^2} \int d^2x \langle h(x)h(0) \rangle e^{-i\mathbf{q}\bullet\mathbf{x}} \quad \text{..... (1.1)}$$

Where h(x) is the height profile of the surface, and q the surface wave vector. This can be directly measured and shows exceptional agreement with a large variety of surfaces.[10] The ideal roughness power spectrum of a self-affine fractal surface is depicted in Figure 1.2, wherein the roughness across all length scales can be described, ranging from the size of the contacting interface down to the atomic dimension. Similar to the more simple models, using this treatment a linear relationship between contact area and applied load at the interface is observed,[9] providing further agreement between the microscale and

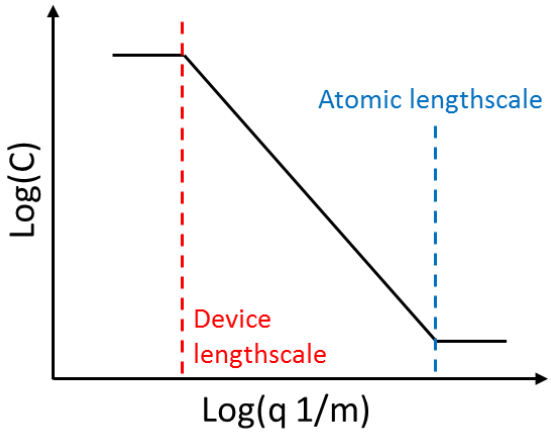


Figure 1.2. The ideal roughness power spectrum of a self-affine fractal surface, showing exponential decay of the roughness magnitude with decreasing length scale q.

macroscale friction laws.

Surface models also provide insight regarding the nature of contact between rough surfaces. For self-affine surface contacts, the population density of contact pressures across the various point contacts at a macroscopic interface is similar to a Boltzmann distribution, where the applied load is analogous to temperature and there exists an exponential tail in the distribution towards greater applied pressures.[11] Similar to a chemical reaction, this tail in the population distribution, representative of single asperity contacts at pressures approaching the mechanical limits of the contacting materials, is responsible for the majority of wear at the interface. In other words, these highest pressure, nanoasperity contacts are where tribochemistry occurs. From a lubrication and surface wear standpoint, the structure of boundary lubricants and surface coatings geometrically confined to the nanoasperity surfaces in these contacts is of greatest

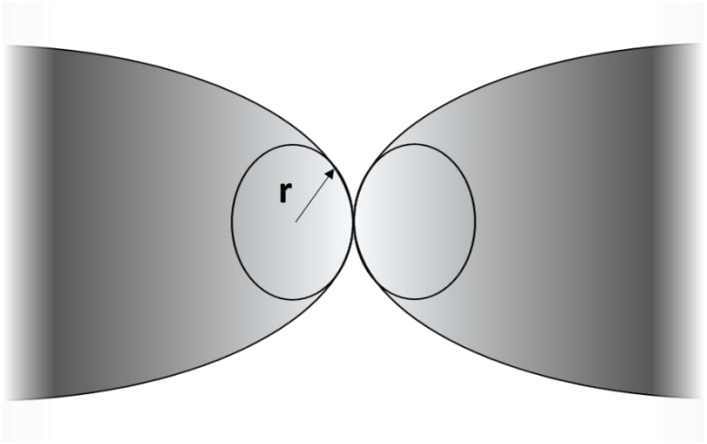


Figure 1.3. Model of a single asperity contact, wherein the point of contact is modeled as contact between two spheres under applied load with a given radius of curvature r.

significance. Thus, an understanding of the asperity contact provides a first step towards understanding how lubricants influence the contact and shear of sliding interfaces.

1.2.3 Single Asperity Contact Mechanics

Several models have been developed which provide an understanding of the single asperity contact. The asperity contact is typically modeled as two spheres in contact, as depicted in Figure 1.3, and the asperity-flat interaction is modeled by assuming one side of the contact has an infinite radius of curvature. The first and most basic theory of asperity contacts referred to as Hertz contact theory,[12] describes contact between asperity surfaces in the absence of surface forces like adhesion. Under this model, the contact radius is determined to be:[13]

$$a = \left(\frac{3NR}{4E^*} \right)^{1/3} \quad \text{..... (1.2)}$$

Where N is the axial compression load, R is the reduced radius of curvature, and E* is the reduced elastic modulus which is often referred to as the contact stiffness, defined as a combination of the Young's Modulus and Poisson Ratio of the materials:

$$E^* = \left(\frac{1-\nu_1^2}{E_1} + \frac{1-\nu_2^2}{E_2} \right)^{-1} \quad \text{..... (1.3)}$$

Where $E_{1,2}$ are the Young's moduli of the two materials, and $\nu_{1,2}$ are the corresponding Poisson Ratios of the materials. Because friction is proportional to contact area, a Hertzian contact would be expected to exhibit a two thirds power dependence on the applied load. The pressure distribution within the contact area follows the parabolic form:

$$p(r) = p_0 \left(1 - r^2 / a^2 \right)^{1/2} \quad \text{..... (1.4)}$$

Where p_0 is the peak pressure at the center of the contact. In the absence of adhesion, there is no interaction outside the contact radius.

Adhesion around the point of contact can influence the pressure distribution in addition to requiring greater repulsive forces from the center of the contact, and improvements to the Hertz contact model largely focus on this issue. Adhesion can arise from long range atomic interactions like van der Waals attractions, or, as is often the case in ambient environments where moisture is typically present, adhesion from the formation of a water meniscus at the contacting surfaces. Most similar to Hertz contact theory is the DMT theory of nanoasperity contacts,[14] wherein it is assumed that adhesion contributes to the overall force of the contact, but is too weak to directly cause material deformations and changes in the pressure and strain distribution. Where the Hertz model predicts zero friction at zero applied load, the DMT model predicts friction forces at zero and negative

applied loads resulting from these adhesive interactions. For surfaces in which the adhesion is strong relative to the mechanical stiffness of the contact surfaces, JKR theory is most applicable.[15] Here, surface adhesion does alter the pressure distribution in the contact, and the contact radius takes the form:[16]

$$a = \left(\frac{4R}{3E^*} \left(N + 3\gamma_{adh}\pi R + \sqrt{6\gamma_{adh}\pi RN + \left(3\gamma_{adh}\pi R\right)^2} \right) \right)^{1/3} \quad \text{..... (1.5)}$$

The factor γ_{adh} is the work of adhesion, representing the energy required to separate the two surfaces absent the deformation energy that is recovered. As this value goes to zero, the definition of the contact radius naturally reverts to the Hertz model. In similar fashion, the pressure distribution is modified:[13]

$$p(r) = p_{Hertz}\left(r\right) + p`_0 \left(1 - r^2 \big/ a^2\right)^{-1/2} \quad \text{..... (1.6)}$$

Where the factor $p`_0$ is defined as:

$$p`_0 = -\left(4\gamma_{adh}E^* / \pi a\right)^{1/2} \quad \text{..... (1.7)}$$

This additional contribution to the contact area increases the interaction area of the two surfaces during contact and sliding and must therefore be considered in the friction response. DMT theory would be most applicable to hard surfaces surface contacts with minimal adhesion, while JKR theory is more applicable to soft surfaces in the presence of greater adhesive forces. Because there is not necessarily a clear distinction between these conditions, however, combination models have also been developed with incorporation of transition parameters that effectively capture the stiffness-adhesion relationship that governs which model is ideal.[17,18]

1.2.4 Lubrication Effects in Asperity Contacts

Surface lubrication represents an additional challenge in understanding the friction response of asperity surfaces. Lubrication is largely defined by the sliding regime under which it operates. Two extremes exist, dry sliding and hydrodynamic lubrication, in which the axial contact pressure is bore entirely by the solid-solid or solid-lubricant interface respectively. Under dry sliding, two parameters are key, the potential energy corrugation of the surface and the mechanical stiffness.[19] The potential corrugation dictates how much restoring force is required to slide the surfaces, and the mechanical stiffness dictates how much strain is required to achieve the necessary sliding force. In hydrodynamic sliding, where solid-solid contact is completely avoided, shear occurs within the lubricant fluid itself. In this case, the fluid viscosity is the primary factor in the friction response. Low viscosity liquids will exhibit the least dissipation between the sliding surfaces, but they are also most easily squeezed out of the contact.[20]

Between these two extremes is the boundary lubrication regime, in which the compressive load is bore by both a lubricant film at the interface and the interface itself. The circumstances under which boundary lubrication occurs are numerous, including contacts lubricated by thin films and surface coatings, when hydrodynamically lubricated surfaces come to rest and the lubricant squeezes out of the contact,[20] or simply as a result of adsorbed contaminants or moisture adsorbed to the contacting surfaces.[21] Even in unlubricated sliding contacts, boundary films are liable to form as a product of surface wear and third body formation, for example, the graphitization of the interface between sliding DLC contacts.[22] Indeed, it has been proposed that in virtually all circumstances,

sliding surfaces undergo some form of boundary lubrication at some point.[23]

Recent applications have further driven the demand for effective and high performing boundary lubricants. While hydrodynamic lubrication is in all cases the most ideal from a friction and wear standpoint because surface contact is completely avoided, not all systems are amenable to liquid lubrication. MEMS devices, microscale machines fabricated typically from silicon, for example, require effective lubrication to operate owing to their high surface area-to-volume ratios, but traditional liquid lubrication is not feasible as it introduces viscous drag that severely inhibits device motion.[24] Space based applications are similarly not amenable to liquid lubrication as sliding occurs in a vacuum, often at extremely variable temperatures where liquid lubricants would be likely to vaporize. A solution in both cases is the development of lasting surface coatings that act as boundary lubricants or which influence the formation of effective tribofilms to mitigate wear at these interfaces. With MEMS, an additional and promising route of lubrication is vapor phase lubrication, wherein the boundary lubricant is constantly replenished by condensation from the vapor phase,[25] though this is still inherently a boundary lubrication problem.

Under boundary lubrication, the mechanical properties of the sliding interfaces contribute to the friction response, but so do the chemical and mechanical properties of the boundary film, which effects the interfacial adhesion and distribution of pressure at the surface contact. A three term friction law is found to be most consistent with the friction response in these circumstances:[26,27]

$$F_f = \tau A + \mu N + F_0 \quad \text{.....(1.8)}$$

Where τ is the interfacial shear strength, μ the coefficient of friction, and F_0 the Derjaguin offset which results from shearing interactions of the boundary lubricant at zero applied load.[28] Increasing lubrication in the contact generally corresponds to decreased interfacial shear strength and contact area, resulting in increasing relevance of the load dependent term. The ubiquity of boundary lubrication in sliding contacts and the load dominant behavior has been used to argue that Amonton's law reflects not a linear correlation between contact area and applied load of statistically rough surfaces, but rather that the load dependent term dominates the friction response of real surfaces owing to boundary lubrication that minimizes the interfacial shear strength.[23]

Unfortunately, the boundary lubrication regime is the most difficult to understand and model using mechanical or continuum techniques. Whereas dry sliding can often be understood in terms of the mechanical properties of the solids, and similarly hydrodynamic lubrication can be described in terms of fluid properties, boundary lubricants are typically so thin that mechanical properties are ill-defined and highly subject to the local conditions at the contact. A sheet of graphitic carbon or a monolayer of metal atoms bares little resemblance mechanically and chemically to their bulk counterparts, and cannot be treated as such, thus ruling out modeling through continuum or mechanical methods. This leaves atomistic MD simulation as the only viable means for modeling the behavior and dynamics of boundary films. From an empirical standpoint, isolation of a boundary lubricated contact requires single asperity contact techniques like AFM and IFM. Unfortunately this limits the types of boundary lubricants that can be studied in a controlled manner, because single asperity contacts offer no restrictions to squeeze out.

Surface coatings which form strong chemical bonds to the surface are therefore ideal, and for this reason self-assembled monolayers have been studied extensively as model boundary lubricants.[29,30]

1.2.5 Self-assembled Monolayers as Model Boundary Lubricants

SAMs are both excellent model systems for boundary lubricants, and have also been successfully employed as boundary lubricant additives and vapor phase lubricants.[31] The SAM, typified by a linker group that binds the molecules to the surface and a structure amenable to organization in 2-dimensions through nearest-neighbor interactions, can be used to alter the surface chemistry of a surface and as a buffer to contact between the interfaces. Though SAMs exist with many different chemical functionalities and properties, the most ideal SAMs for surface lubrication are saturated hydro- and fluorocarbon SAMs. The stability of C-H and C-F bonds imparts a lower energy to otherwise more reactive surfaces, minimizing interfacial interactions that contribute to greater adhesion and mechanical coupling of the surfaces during sliding.

Thiolate and silane derived SAMs are the most common, and can be applied to metal and metal oxide surfaces respectively. Though the only obvious difference between these two types of SAMs is the linker group, they are different in many ways. Thiolate SAMs may bind directly to a metal surface via a relatively labile metal-S bond. Silane derived SAMs form irreversible bonds to metal-oxide surfaces at dangling oxygen atom sites, and may also form bonds with one another or, alternatively, may interact via hydrogen bonds with the surface, nearest neighbor molecules, as well as water molecules intercalated between the surface and the film. The considerable variability in silane chemistry renders

it much more sensitive to assembly conditions and surface morphology,[32,33] but it also means that surface chemistry is not critical to SAM formation,[34] providing greater versatility. Silane SAMs can even be formed on substrates which are largely devoid of direct binding sites, owing to the large variety of interactions involved in their assembly.[35]

Despite their differences, SAMs of thiolates and silanes have both been extensively studied as model lubricants, and silane SAMs in particular have been considered as friction and wear reducing layers in silicon based MEMS.[36] Not only do SAMs provide chemical passivation of the sliding interface, they also offer reversible dissipation pathways that include tilt deformations and conformational changes in the molecules structure, dissipation pathways which do not contribute to wear of the interface.[37] As lubricant additives, their strong binding and cohesive energies further aid in maintaining their presence in surface contacts even when the lubricant fluid has been completely squeezed out.

Salmeron et al examined the friction response of OTS SAMs on mica from zero load to sufficient load to wear the underlying surface,[38] observing four primary regimes of friction response for these model lubricants, depicted in Figure 1.4. At the lowest loads, corresponding to wear-free lubrication of the contact, a linear relationship between friction and load was observed. In the second regime, an increasing friction coefficient was observed and attributed to increasing distortion and displacement of the SAM. The third and fourth regimes correspond to the tip achieving contact with the substrate and directly wearing the substrate respectively. This variation in the friction response indicates perhaps most importantly that in the boundary lubrication regime, the interfacial shear

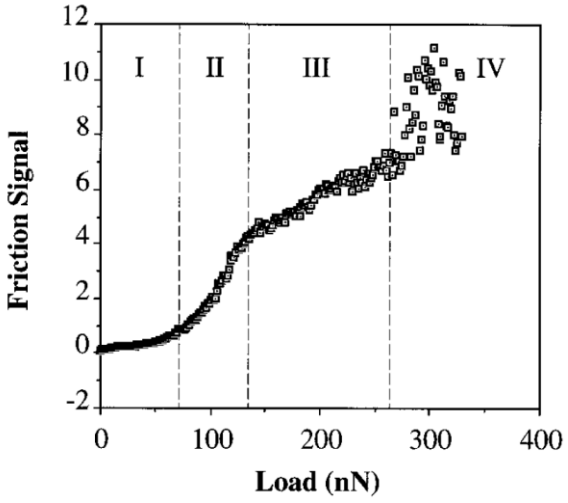


Figure 1.4. Friction response of OTS measured on mica, exhibiting four characteristic regimes ranging from wear-free lubrication (I) to wear of the underlying surface (IV). Reproduced with permission from Xiao, X.; Hu, J.; Charych, D. H.; Salmeron, M. "Chain Length Dependence of the Frictional Properties of Alkylsilane Molecules Self-Assembled on Mica Studied by Atomic Force Microscopy". *Langmuir*. **1996**, 12, 235-237.[38] Copyright 1996 American Chemical Society.

strength and coefficient of friction depend on the applied load, and understanding this variation is critical to understanding the action of boundary lubricants.

The extreme loads used to explore all the frictional regimes of the OTS SAM are not characteristics of the vast majority of works examining SAM friction, with most studies being primarily in the linear regime. For example, Leggett and coworkers have extensively examined the effects of interfacial forces and environment on the friction response of SAMs. In particular, they observed that the contact area dependence of SAMs as lubricants is recovered when the interaction forces between opposing SAMs are increased, as depicted in Figure 1.5. Through variation of SAM head groups and solvent environments, they observed a correlation between surface adhesion and the impact of

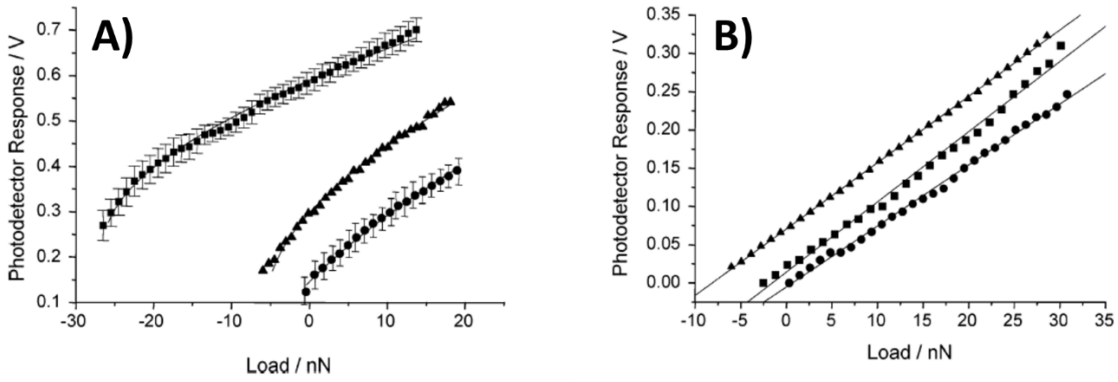


Figure 1.5. Friction in PFD (A) and ethanol (B) for 11-MUA SAMs on Au(111) with a silicon nitride probe (triangles) or a gold coated tip functionalized with 11-MUA (squares) or dodecanethiol (circles). The strongest contact area dependence was observed for the COOH mated contact in PFD, indicated by negative curvature. When measured in ethanol, only linear relationships between friction and load are observed due to interference of the intermolecular interactions with the solvent. Reproduced from Colburn, T. J.; Leggett, G. J. "Influence of Solvent Environment and Tip Chemistry on the Contact Mechanics of Tip−Sample Interactions in Friction Force Microscopy of Self-Assembled Monolayers of Mercaptoundecanoic Acid and Dodecanethiol". *Langmuir*. **2007**, 23, 4959-4964.[39] Copyright 2007 American Chemical Society.

contact area on the sliding contact.[39-41] This belies the primary effect of the boundary lubricant, which is to mechanically decouple the sliding interface. The mechanical coupling of the sliding interfaces is expressed in terms of the interfacial shear strength, the coefficient of associated with the contact area dependence. By increasing the interaction between the sliding interfaces, they become more mechanically coupled and an increased mechanical contribution to the friction response is observed.

Though much can be learned from AFM measurements of boundary lubricants on flat surfaces, however, in real contacts, particularly at the high pressure contacts that dominate wear of the interface, contact occurs between two asperities. From a continuum modeling

perspective, this is no problem, an asperity-flat contact can be directly mapped to an asperity-asperity contact since the pressure distribution of the interface depends on the reduced radius of curvature. In the boundary lubrication regime, however, this is not necessarily the case. SAMs, for example, rely on cohesive interactions both for assembly and to improve their mechanical fortitude, and surface curvature on molecular length scales can disrupt these interactions. Similarly, surface curvature induces strains in two dimensional materials like graphene and molybdenum disulfide that increases surface reactivity and promotes wear. It is therefore essential to understand how the localized factor of surface curvature influences the structural, dissipative, and passivating properties of boundary lubricants to better optimize their design for demanding lubrication applications.

1.2.6 Introduction to Subsequent Chapters

The effects of surface curvature on the dissipative properties of SAMs were determined for OTS SAMs on silica nanoasperity and flat surfaces using atomistic MD simulation. In addition, because silane SAM formation is affected by local surface curvature with variations ranging from 33-100% of a full monolayer,[32,33] the effects of packing density were also considered. SAMs as silica surface coatings offer two key benefits. First, they reduce the surface energy of the otherwise reactive silica surface. Second, they offer reversible dissipation pathways through tilt deformations and the formation of conformational defects. Thus, the SAMs were evaluated for their effective surface coverage, an indicator of passivating benefit, as well as the density of gauche defects and tilt away from the surface, which address their potential to dissipate sliding

energy reversibly, as a function of surface curvature and packing density.

Of these two variables, surface curvature and packing density, it was found that packing density was the most determinative factor in the properties of the SAM surface coating, and the relationship between boundary film density and the contact mechanics of asperity interactions were thus considered. This was achieved by pressing the functionalized surfaces into contact, and measuring the distribution of pressure and strain in the contact plane. To achieve this, the atomic forces, positions, and potential energies were combined to produce two-dimensional pressure maps. Best fit Hertzian pressure distribution models were used to compare the resulting pressure distributions against one another and also against the continuum model. In addition to understanding the relationship of boundary film density and contact mechanics, this work also elucidates the origins of the multi-regime friction response, explicitly demonstrating a sharp transition in the strain localization in the film that gives rise to the observed transition from a linear friction response to a friction response with increasing friction coefficient.

In Chapter II, the simulation techniques used in this work are explained in Section 2.1, and the methodology for generating the SAM functionalized silica substrates is explained in Section 2.2. In Chapter III, the effects of surface curvature and packing density on the structure of SAMs was determined by examining equilibrated structures of SAMs on silica nanoparticles and on flat surfaces. The development of methodologies for analyzing pressure distributions in atomically simulated contacts is discussed in Chapter IV, and the relationship between film density and contact geometry on the redistribution of pressure is explored. These methods were additionally applied to better understand the friction

response of SAM measured experimentally, and this is described in Chapter V.

1.3 Confinement Effects in Molecular Electronics

1.3.1 Molecules as Functional Electronic Devices

The desire to use molecules as functional electronic devices stems largely from the desire to continue decreasing the size of electronic components on semiconductor devices. With decreased size comes increased device density that imparts greater processing power to microchips, greater storage density to optical, magnetic, and solid-state media, and greater resolution in devices like CCD cameras. Additional benefits include faster response times, equating to greater computing power, and lower voltage requirements meaning reduced temperatures and cooling requirements. These trends conspire to prevent the energy demands of an ever-growing computing industry from scaling directly with its capabilities. Unfortunately, size reduction is reaching its limits as a tool to increase computing power and efficiency. The solid-state device framework is beginning to erode as devices become so small, on the order of 10-20 nm or smaller, that their size, in addition to their local environment, begin to have more profound effects on their function. It necessarily becomes more necessary to think of these devices as molecular systems, and this presents the alternative notion, can molecules be used as devices?

The modern conception of a molecular electronic device was originally conceived by Aviram and Ratner,[42] with the notion that a donor-acceptor unit could rectify current across a junction. It was some time before the appropriate instrumentation was developed to investigate this hypothesis, but it has indeed been observed that current rectification can be achieved across a molecular junction.[43] Rectification is perhaps the most important

function a molecule can serve, as this provides for control of charge flow across an interface, a feature not only essential in devices but also an important to increasing the efficiency of dye-sensitized photovoltaics, in which recombination inefficiencies can be avoided with sufficiently rectifying organic dyes systems. Molecules could also potentially serve as insulators, conductors, and transistors, or with capabilities that exceed traditional semiconductor devices. Broadly speaking, molecules with highly localized molecular orbitals like saturated hydrocarbons have insulating properties, while molecules with highly delocalized orbitals via conjugation and extended aromaticity can serve as molecule wires. Owing to the discreteness of states in molecular systems, however, their behavior is highly bias dependent, and NDR is a purported feature that is more unique to these and other nanoscale systems,[44-46] wherein current flow exhibits a maximum at a specific bias.

In addition to the benefits stemming from the size and potential function of molecular devices, their mode of design and fabrication is completely different from traditional technologies. Solid-state devices are prepared by top-down methods which, though there have been great strides in fabrication techniques, ultimately limits throughput. Molecules can be made in mole quantities, and they can be synthetically tailored with nearly limitless combinations of chemical functionalities that can yield highly specific conductance responses to external stimuli and gating effects. The challenge is ultimately not one of device fabrication, but device incorporation, and this is a particularly immense challenge even in a laboratory settings, much less a production setting. The sensitivity of molecular conductance to factors like molecular orientation,[47], conformation, [48,49] surface binding

geometry,[50] and the local surface chemistry and chemical environment[51,52] requires that all of these features be consistently controlled.

There are therefore two essential challenges that must be addressed for molecules to effectively be used to control charge flow across at interfaces and to serve as electronic devices. First, the means to reliably fabricate molecular junctions is essential. This does not mean simply forming junctions that do not short, but junctions which are completely uniform in terms of the local environment of the molecular device. Second, a complete understanding of the relationship between molecular structure and charge transport is essential. This requires an understanding of the modes of molecular conduction, and how these modes of conduction can be manipulated. Finally, these are not isolated problems. The nature of the junction will influence molecular conduction, and understanding this more complex relationship is critical.

1.3.2 Characterizing Molecular Charge Transport Junctions

A variety of methods exist to measure the conductance of tunnel junctions,[53] but, because local environment matters, technique matters. The available techniques can be broadly categorized between large area junctions and single molecule junctions. The general benefit of large area junctions is that they sample a larger number of molecules, providing improved signal-to-noise ratios by averaging over the various local conditions that can give rise to variation in the response of a single molecule. The drawbacks, however, are numerous. Traditional CVD deposition of electrodes tends to result in filament formation, and the condensation of the hot metal atoms atop the organic layer has been shown to chemically modify the molecular junction.[54] Transfer printing provides a

softer method of electrode placement wherein the electrodes are prefabricated and placed in contact with the film,[55] however this approach introduces challenges of contact conformity, and nevertheless filament formation and other dynamics with the application of bias have been observed even after the fabrication step.[56] The primary difficulty then is ensuring that the behaviors observed are intrinsic to the molecular junction and not a property of or contributed to by the deposited electrodes. Liquid metal electrodes of Hg[57] and eGaIn[58] have shown greater promise in this regard, being both easier to use and more reliable, though their use is limited to laboratory settings.

Of the single molecule techniques, STM is perhaps the most versatile, allowing for simultaneous imaging and measurement of the molecular junction. Additionally, STM does not mechanically or chemically perturb the contact, as a vacuum gap exists between the tip and the molecule. This allows in particular for characterization of molecules that need only bind to the substrate, and though it does introduce asymmetry in the contact, this is an effect which can nominally be controlled for. Similar to STM, CP-AFM provides a relatively highly localized measure of molecular conductance for junctions in mechanical contact, thereby eliminating the vacuum gap, though contacts do not necessarily consist of single molecules.[59] Single molecule techniques in which the molecules bind both electrodes include break-junction[60] and electromigration[60] techniques, the former being easily achieved by approaching and retracting an STM or CP-AFM tip to a functionalized surface.

A key difference that is often neglected between these methods is the influence of the junction geometry on the molecular junction. A molecular junction is often considered

primarily in terms of conductance, or conversely, resistance, but due to the close proximity of the electrodes, metal-molecule-metal junctions may also be viewed as capacitors, with the molecular layer acting as a very thin dielectric. With a single molecule junction, where the current flow is localized to an asperity almost in contact, the electric field resulting from the applied bias is much more focused upon the tip-substrate junction.[61] The high localization of the electric field in STM has been used, for example to pattern atoms[61,62] and ablate molecules from the surface.[63] More importantly, the localized electric field can induce non-destructive changes in the molecular junction, potentially polarizing molecular orbitals so as to alter their energetics or causing changes in conformation of the junction that must be considered.

In addition to these effects, the difference between a single molecule junction and a large area junction is the relevance and nature of intermolecular interactions, which can include electronic coupling effects[64] and steric hindrances that limit motion and conformational changes of the molecule.[65] One would not expect a stark difference between an STM measurement of current flow through an alkanethiol and a large area junction, because the interactions between the molecules are unlikely to have a strong influence on their electronic structure. Alternatively, aromatic molecules like OPEs and porphyrins can interact with greater degrees of electronic coupling, and one would expect differences in large area junctions and isolated single molecule measurements as a result. Transport in organic thin films is similarly disconnected from single molecule conductance owing to the formation of molecular domains and domain boundaries through which charge must flow Thus, the degree the local environment can effect molecular

conductance also depends on the susceptibility of the molecule to environmental effects, and the strength of interactions between nearest-neighbors. These effects must be considered when comparing the conductivity of molecular thin films, large area molecular junctions, and single molecule junctions.

1.3.3 Mechanisms of Charge Transport in Molecular Systems

Molecular conduction can most simply be viewed as a sort of chemical reaction:[66]

$$D^- - M - A \longrightarrow D - M - A^- \quad \text{..... (1.9)}$$

Where D is the donor or source electrode, and A is the acceptor or drain electrode, and M represents the molecule in the junction. The mechanism of charge transfer, however, largely depends on the structure. The most critical parameters are the orbital and charging energies of the molecular junction, and the coupling between the junction and the electrodes. The two primary modes of conduction are superexchange, often referred to as off-resonant or resonant tunneling, and sequential charge transfer, often referred to as charge hopping.[66] Superexchange describes an electron transitioning from a source state to a drain state, and is unique to molecular junctions largely due to their extremely short separation distance, which allows significant electrode state density to project through the junction. This is not observed for wider junctions because of the strong distance dependence of superexchange, which is exponentially dependent on gap width:

$$I \propto e^{-\beta z} \quad \text{..... (1.10)}$$

Where β is termed the tunneling efficiency. Sequential charge transfer is less distance dependent and is dominant in materials like conductive polymers,[67] organic thin films,[68] and long molecular junctions.[69]

In terms of the simple reaction depicted in Equation 1.9, the intermediate states indicate the key difference between these mechanisms. In superexchange, there is no intermediate, the electron exchanges between source and drain states and is never localized on the molecular junction, and this is possible because the states of the two electrodes are not completely isolated from one another. In sequential transfer, the charge resides on the junction transiently as it passes from one electrode to another, and could nominally be viewed as multiple superexchange processes.

Examples of molecules which exhibit superexchange transport are hydrocarbons like

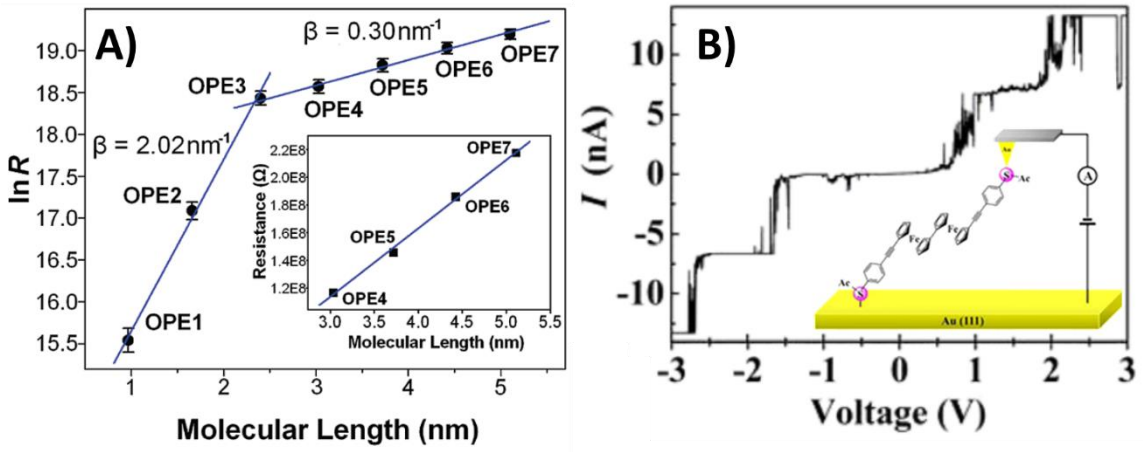


Figure 1.6. Characteristics of sequential charge transport indicated in (A) by a marked decrease in the tunneling efficiency with increasing length for OPE molecular wires, and in (B) by the observation of Coulomb blockade, wherein the sharp increases in conductivity arise as molecular charge states enter the bias window. (A) reproduced with permission from *ACS Nano*, 3, 3861-3868.[70] Copyright 2009 American Chemical Society. (B) reproduced with permission from Chen, C.-P.; Luo, W.-R.; Chen, C.-N.; Wu, S.-M.; Hsieh, S.; Chiang, C.-M.; Dong, T.-Y. "Redox-Active π-Conjugated Organometallic Monolayers: Pronounced Coulomb Blockade Characteristic at Room Temperature". *Langmuir*. **2013**, 29, 3106-3115.[71] Copyright 2013 American Chemical Society.

alkanethiols, by which the primary mode of transport is through-bond tunneling, [72] as well as short molecular wires, wherein mixing of molecular and electrode states offers greater coupling between the electrodes and improved transport efficiency. Hopping based transport can be a result of increased molecular length, wherein superexchange is suppressed due to its strong distance dependence, or by decoupling superexchange transport pathways by reducing the broadening and delocalization of electronically active functional groups from the electrodes, which effectively isolates the molecular states.

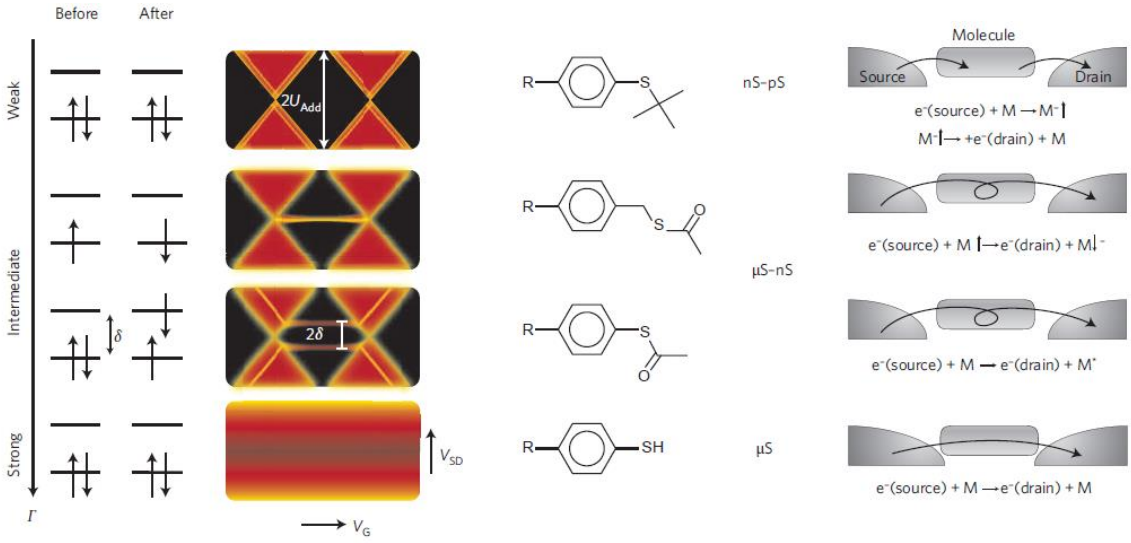


Figure 1.7. Mechanistic dependence of charge transport on the electronic coupling between the molecule and the electrode surface. In the above examples, only the linker group is changed, with R representing the rest of the OPV molecule. With increasing decoupling of the molecule from the surface, more distinct charge bearing characteristics are exhibited as indicated by the second column plots of conductance vs. V_G and V_{SD}. Reproduced by permission from Macmillan Publishers Ltd: *Nature Nanotechnology* Moth-Poulsen, K.; Bjornholm, T. "Molecular electronics with single molecules in solid-state devices". *Nat. Nanotechnol*. **2009**, 4, 551-556,[73] copyright 2009.

Because sequential transport is less distance dependent, it can be identified often as a change in the transport efficiency of a junction with length,[70,74,75] for example that observed for OPE molecules depicted in Figure 1.6A. Sequential transport may also be identified by the observation of junction charging effects like Coulomb blockade,[76,77] as depicted in Figure 1.6B for a junction based on ferrocene activity.[71] Because sequential transport relies on charge states of the molecular junction, junctions of this type are also subject to greater control through the application of gating bias. Figure 1.7 for example depicts the gate bias dependence of OPV molecules depending on the linker group used to bind them to the electrode.[73] With decreased coupling, it is observed that gate bias dependence increases dramatically, owing to a change in transport mechanism.

The mechanism of transport in a junction largely dictates the capabilities and expectations one may have. For superexchange, the tunneling efficiency β is the determining parameter that can be varied. Tunneling efficiencies ranging from ~0.5 $Å^{-1}$ for aromatic molecules to as high as 2.3 $Å^{-1}$ for a vacuum gap[78] can be achieved, yielding 7 orders of magnitude difference in current flow for a 1 nm junction. This is substantial, but it is unreasonable to expect that a molecular junction could be conceived that dynamically and reversibly switches between something and nothing. Assuming that there must exist some molecular species in the gap limits the upper range of tunneling efficiencies to that of a hydrocarbon, ~ 0.8 - 1.2 $Å^{-1}$, producing a more modest 100-fold difference in junction conductance. This presumes that the tunneling efficiency of the entire junction is made to vary in response to some stimulus, which is similarly unlikely, so the actual variation in conductance that could be achieved in a tunnel junction would

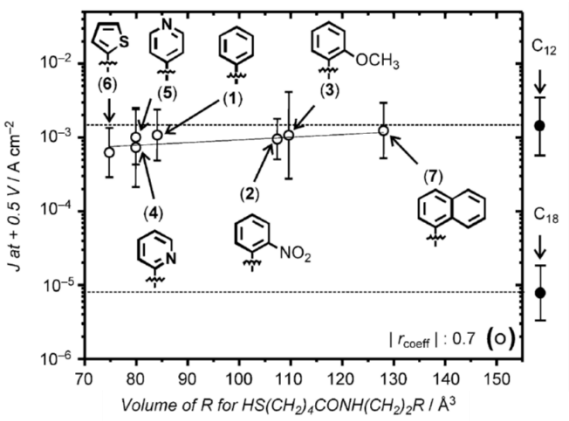


Figure 1.8. Variation in junction conductance of SAMs with different terminal groups studied using an eGaIn top junction contact. The variation in conductivity of the molecular junctions was found to not deviate far from what would otherwise be predicted just by determination of the molecular length. Reproduced with permission from Yoon, H. J.; Shapiro, N. D.; Park, K. M.; Thuo, M. M.; Soh, S.; Whitesides, G. M. "The Rate of Charge Tunneling through Self-Assembled Monolayers Is Insensitive to Many Functional Group Substitutions". *Angew. Chem. Int. Ed.* **2012**, 51, 4658-4661.[79] Copyright 2012, Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim.

be much less.

This point has been borne out through extensive research by Whitesides and coworkers, in which substantial variation of molecular structure was employed to investigate the relationship between chemical structure and junction conductivity. Works considering the effect of terminal groups,[80] internal molecular structure,[81] and even complete variation of up to half of the molecular junction,[79] indicated negligible changes in conductance that could largely be attributed to variations in molecular length, not the more detailed structure and electronic characteristics of the molecular junction, and example of these studies are depicted in Figure 1.8. Similarly, they and others have found that through tunnel junction asymmetry, a means of achieving current rectification,

rectification ratios of a paltry 20 are the most that can be reasonably achieved,[43,82] orders of magnitude lower than current solid-state technologies. In all cases, the impact of variations to the tunneling efficiency can be magnified by increasing the length of the junction, but this leads to dramatic reductions in their overall conductivity, a trade-off which limits the junction length to a few nm.

With superexchange based transport limited essentially to molecular insulators and wires, devices that require little dynamic variation in conductivity, the benefits of achieving sequential charge transport in junctions are clear. The transmission states, corresponding to charging states of the molecule, can be more directly controlled through synthetic design, offering better control over junction response. Response to a gate bias also imparts greater functional characteristics that can be extended to include electrochemical control, and chemical control or response to optical stimuli impart sensing capabilities. Furthermore, particularly in the case of electronically decoupled molecular junctions, junction charging is indicative of the ability for the molecule to confine charge, a feature which has been used in molecular storage applications.[83] Assuming a molecular footprint of ~2 nm, molecular data storage could correspond to a 100-fold improvement in data storage density over current solid-state storage technology. The ability to effectively stabilize charge near the electrode interface is also important in dye sensitized photovoltaics, wherein recombination inefficiencies can be reduced through a more stabilized charged state of the dye molecule prior to hole extraction.

Control over coupling between electronically active subunits, as well as stabilization of potential charge states, is key. The formation of charged molecules must take several

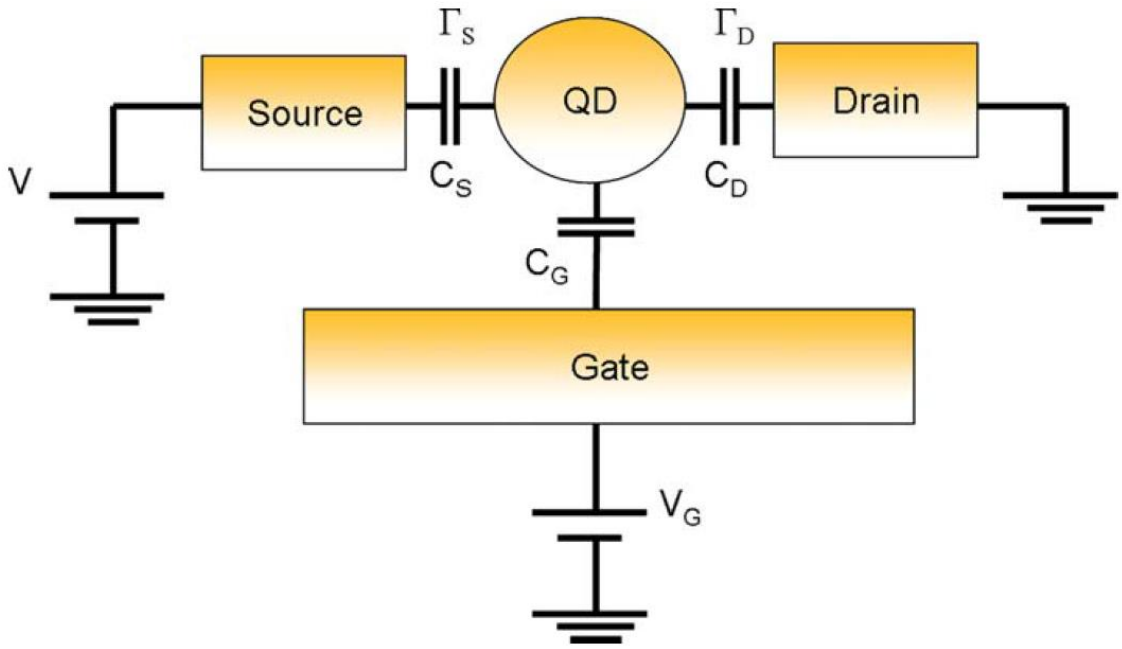


Figure 1.9. Configuration of an electronically active molecular component, described generically as a quantum dot (QD), capacitively coupled to a three-electrode system. The capacitance of the QD, dictated by C_S and C_D, as well as the coupling of the feature to the source and drain electrodes, Γ_S and Γ_D, dictate the ability of the feature to store charge, and thereby support sequential transport. Reproduced from Selzer, Y.; Allara, D. L. "Single-Molecule Electrical Junctions". *Annu. Rev. Phys. Chem.* **2006**, 57, 593-623.[84] Copyright 2006 Annual Reviews.

factors into account, in particular, the ability of the molecule to delocalize charge in order to minimize Coulomb repulsions and the quantum confinement energy,[84] and the influence of the local environment to stabilize charge. Electrode coupling plays a significant role in these parameters as well, as electrode coupling can serve to delocalize molecular charge states into the electrodes, and image charges in the electrode can serve to stabilize charge states of molecules. Allara et al describe the necessary conditions to stabilize charge in a molecular junction,[84] wherein the electronically active functional group may be viewed as

a feature capacitively coupled to the system electrodes, this is depicted in Figure 1.9. To effectively stabilize charge, the capacitance of the active feature must be great enough to support at least one electron at the desired bias. The capacitance of this charged island depends on its size,[77] as well as the distance and dielectric environment between the feature and the electrodes.[85] Control over these features thereby provides a means of mechanistic control of charge transport in molecular junctions.

1.3.4 Introduction to Subsequent Chapters

Two dimensional nanoconfinement was employed as a means to vary the mechanism of charge transport in a hydrocarbon tethered zinc porphyrin molecule on the Au(111) surface. The freebase analogue of this molecule has been previously characterized, exhibiting transport via tunneling owing to the decoupling effect of the hydrocarbon tether. This molecule has been studied by mixed self-assembly, and virtually identical conductivity characteristics were observed for isolated molecules on the surface, however, the addition of zinc promote increased molecular aggregation, and the observation of more conductive molecular islands as well as Coulomb blockade characteristics in crossed-wire tunnel junctions. It was therefore hypothesized that the increased conductivity and charging characteristics were a result of molecular aggregation, and that there should exist a direct relationship between aggregate size and structure conductivity.

To confirm this hypothesis, isolated control over feature size via two dimensional nanoconfinement was employed. The AFM nanografting technique was used to promote directed assembly of these porphyrin islands with specified fabrication dimensions. The goal of this work was to direct control over the transport mechanism by varying the size

of the electronically active feature while maintaining the same coupling between the feature and the electrodes, thereby offering a new route to the formation of tunable and electronically active molecular junctions. Optimization of this technique and verification of structural quality are discussed, and the relationship between aggregate size, conductivity, and transport mechanism are explored in Chapter VI.

CHAPTER II

EXPERIMENTAL METHODS

2.1 Classical Molecular Dynamics Simulation

Classical molecular dynamics simulation is a technique for computational modeling of compounds and systems that has been employed to model an enormous variety of systems.[86,87] In this simulation technique, empirical force fields are used to describe the interaction forces between atoms, and Newton's equations of motion are used to propagate the atoms' positions in time in response to these forces. The most beneficial aspect of this technique, compared to more complex *ab initio* and density functional theory approaches, is that it scales linearly with system size due to the relatively simplicity of the computations involved. Furthermore, it is easily scalable and parallelized, with the commonly used LAMMPS software package reporting benchmark simulations of billions of atomic particles spread across many thousands of processors.[88,89] Because of this, molecular dynamics is particularly well suited to solid and liquid state systems that may otherwise be difficult to examine with higher level calculations due to their high particle density, as well as large biomolecules like proteins and DNA.

Broadly speaking, the simplicity of these calculations does come with a cost. To accurately simulate a system, the force fields that define the interatomic interactions should be highly specialized to model the system of interest, for example it is unlikely that the bulk properties of amorphous carbon could be accurately deduced using a force field that has been defined against the properties of organic molecules. Furthermore, the

formation and breakage of chemical bonds is extremely challenging to model because many atomic interactions depend highly on their local environment and can be highly anisotropic, and capturing this behavior in empirical force fields can be very challenging. Highly complex force fields with many-body interactions have been developed that address these issues,[90,91] but with greater complexity comes greater computational cost, limiting the size of simulation models and the length of time their dynamics may be propagated and examined. Quantum molecular dynamics represents the most extreme case, in which extremely high computational cost yields the highest achievable accuracy, but these calculations are subject to the same restrictions typical of quantum calculations and timescales on the order of femtoseconds.

An additional and important area of limitation for molecular dynamics simulation is that of timescale. Though computations can be spatially parallelized relatively efficiently, parallelization in time is generally not an option. Parallel replica dynamics[92] may be used for infrequent, but quick events, in which several parallel, perturbed replicas of the simulation model are propagated in parallel, but this is not a solution for processes that simply take a long time (μs or greater). The general constraint on timescale stems from the size of the time step that must be employed. In a fully atomistic simulation involving hydrogen atoms, for example, a time step of a fraction of a femtosecond must be employed to properly replicate the hydrogen atom vibrations on the order of 10^{14} Hz. The only solution to achieving greater timescales is to limit fast timescale motion. United atom models in which the hydrogen atoms are treated as part of their bonded substituent[93] or algorithms designed to eliminate hydrogen atom motion[94] can yield a 10-fold increase in

time step, simply because most other atoms are at least 10 times heavier, corresponding to a 10-fold increase in simulation time. Further coarse-graining has been performed with polymeric systems[95,96] and proteins,[97] which can result in even greater simulation timescales at a loss of simulation detail and potentially accuracy.

Tribological simulations are subject to many of the trade-offs previously discussed. Many varieties of tribological simulations exist, including simulations to understand the structure of surfaces and coatings,[98-101] equilibrium properties of surface contacts,[102-105] and dynamic behavior of sliding and contacting interfaces.[106-108] When studying the interactions between surfaces, the complexity of the force fields is often a critical parameter. Interactions can be greatly influenced by the chemistry at the interface, so force fields that can suitably simulate the chemistry of the interface, if available, represent the best choice. Unfortunately, the timescales of macroscopic motion of sliding and contact interfaces are also quite large, so the simplest force field that can accurately model the system is ideal. The sliding speeds of devices span many timescales, many much slower than can be reasonably simulated. A cylinder in an automotive engine, for example, moves at speeds ranging from 10-20 m/s, interfaces in MEMS devices slide at rates ranging from μm/s to m/s, and AFM measurements, which actually represent the most simple and well-characterized surface contacts, exhibit sliding in the nm/s to μm/s regime. Considered in terms of 100 ns of sliding, a relatively long time for a fully atomistic simulation, an AFM tip traverses one millionth to one thousandth of an Ångstrom. At a more favorable speed of 1 m/s, the sliding interface would traverse a more favorable 100 nm, which is usually sufficient to reach a steady state from which

measurements can be extracted.. An alternative approach, which is employed in this work to model AFM measurements, is to assume that the sliding speed is so slow that the contact does not deviate far from equilibrium, such that static contact conditions are employed and measurements are collected at equilibrium.

2.1.1 The Molecular Dynamics Simulation Method

In the simplest sense, molecular dynamics simulation is a numerical solution to a complex, second order differential equation. The initial condition consists of the geometry of the atoms in the system, and the differential equations dictating the system behavior are the definition of the forces acting between the atoms, also known as the force field. Integration of these forces with respect to time yields the particle velocities, and the second integration yields their positions. Therefore, to successfully conduct a molecular dynamics simulation, one need only determine a reasonable initial condition and apply an appropriate force field. Without the inclusion of any external constraints like thermostats, barostats, external forces or fields or moving features, the numerical solution to these equations represents a microcanonical ensemble in which the number of species, the volume, and the energy remain fixed. The solution will therefore explore the state space determined by these variables.

While the initial conditions are highly specific to the sorts of systems and dynamics being investigated, the numerical integration of the atomic forces is generally more straightforward, with a few options available. In the work described herein, two primary algorithms are employed, the widely used velocity-Verlet algorithm, and the RESPA integration scheme. The primary goal of the integrator is to accurately propagate the

particles in time, which in general means that energy must be conserved for the largest possible number of time steps chosen. While a numerical solution will inevitably deviate substantially from an analytical solution, as long as the total energy of the system is unaffected the state of the ensemble is preserved. The general formulas for the velocity-Verlet algorithm are:[109]

$$x(t+\delta t) = x(t) + \delta t\dot{x}(t) + \tfrac{1}{2}\delta t^2\ddot{x}(t) \quad \text{..... (2.1)}$$

$$\dot{x}(t+\delta t) = \dot{x}(t) + \tfrac{1}{2}\delta t\left[\ddot{x}(t) + \ddot{x}(t+\delta t)\right] \quad \text{..... (2.2)}$$

The most complex calculation, determination of $\ddot{x}$ from the atomic coordinates and force field description, need only be computed one time, and both the position and velocity of the particles are factored into the computation, incorporating a brief history of the particle motion in determination of the propagated position and velocity. More complex integration routines that incorporate more of the particle's history to compute its behavior may offer slightly improved adherence to the analytical solution, but this is simply not a priority compared to maintaining the efficiency of this core computation. The RESPA integration scheme is a multi-timescale integrator that works in similar fashion to the velocity-Verlet algorithm, except that different interactions are computed with different frequencies.[110] Because the magnitude of forces for different interactions like bond stretches, angle bends, torsional and long range interactions are often dissimilar, this approach can be employed to achieve a larger time step by considering the stronger interactions, which give rise to more rapid changes in position and velocity, more frequently. A typical scheme, which is employed in this work, is to consider the bonded

interactions every time step, the weaker angle-bending interactions every other time step, and the weakest torsion and long range interactions every fourth time step. This latter point can be important, long range interactions between particles are generally weak, except for systems under pressure, where particles in close contact are actually subject to the quickly varying repulsive portion of the van der Waals potential. For this reason, the RESPA integrator was chosen to simulate the dynamics of the free-standing surface coatings discussed in Chapter III, while the velocity-Verlet integration scheme was chosen for simulating the coatings in compressive contacts discussed in Chapters IV and V.

2.1.2 Temperature Control of Simulations

In many cases, a simulation may be conducted at constant energy, the so-called "Microcanonical ensemble". This can nominally be achieved by simulating the system with no external impetus, and the sum of potential and kinetic energy will remain fixed within the numerical accuracy of the integration technique. When simulating real systems under dynamic conditions, however, it is often more appropriate to perform the simulation with a constant temperature, the "Canonical ensemble". This is achieved by adding or removing kinetic energy to the particles in the simulation, and can be effectively viewed as performing the simulation in an energy bath, for which energy is conserved for the combination of system and bath, and temperature of the system is controlled by exchanging energy with the bath. This is, in effect, how the commonly employed Nose-Hoover thermostat[111] operates when applied to simulations. The key defining term then is the coupling constant, which dictates how quickly energy moves between the bath and the real degrees of freedom of the system, which must be defined carefully so that the

thermostat does not overcompensate fluctuations, but does not too slowly push the system into thermal equilibrium with the bath.

Velocity scaling algorithms are also effective in controlling the temperature in a more simplistic fashion. This is the basis for the Langevin thermostat[112] employed in this work The temperature of a system is defined as:

$$T = \frac{1}{3Nk_B}\sum_i m_i \left|\mathbf{v}_i\right|^2 \quad \text{..... (2.3)}$$

In order to exert control over the temperature of the particles, the atomic forces may be modified to include terms which depend on their velocity:

$$m\ddot{x} = F_c - m\dot{x}\gamma + R(T) \quad \text{..... (2.4)}$$

Where F_c is the conservative force that would apply in the absence of a thermostat. The second term defines a frictional force acting on the particles depending on their velocity, effectively damping atomic motions with a strength dictated by γ. The final term, R(T), is a function that applies fluctuations to the kinetic energy adjustments of the particles. For the Langevin thermostat, R(T) is defined as:

$$R(T) \propto \sqrt{\frac{k_B T m}{\gamma dt}} \quad \text{..... (2.5)}$$

This type of thermostat may best be described as performing the simulation in a fluid bath. The damping term of Eq. 2.4 represents the viscosity of this fluid bath, and the R(T) term represents thermal fluctuations in the fluid bath itself.

2.1.3 Force Fields for Molecular Dynamics Simulation

The most important decision to be made when employing molecular dynamics

simulation is the choice of force field. This choice dictates the complexity of the simulation model in terms of both resolution and the complexity of the dynamics considered. The force field consists of the definition of the potential energy of the interactions between the particles in the simulation. Based on this definition, the forces can be derived from the simple equation:

$$m\ddot{x} = \mathbf{F}(\mathbf{x}) = -\frac{\partial U}{\partial \mathbf{x}} \quad \text{..... (2.6)}$$

There are many classes of force fields available, which can broadly be classified as: two-body, many-body, and those with defined molecular topologies. Two-body force fields are convenient and efficient, wherein the force between two atoms depends only on the distance between them. This therefore presumes isotropic interaction and that the interaction does not depend on the presence of other nearest neighbors. Examples of systems suitable to this kind of force field are ionic crystals and a noble gases, where interactions are primarily Coulombic and van der Waals respectively, though these force fields can be optimized to reasonably match the structure of covalently bound solids. For example, the distribution of oxygen atoms around silicon atoms in SiO_2 may be driven by the Coulombic repulsion of the negatively charged oxygen atoms around the positively charged silicon center, which will result in maximal separation of the oxygen atoms and resulting O-Si-O bond angles consistent with SiO_2 without the use of 3-body interactions to enforce proper angles. Many-body force fields range from relatively simple systems restricted to 3-body interactions,[113,114] embedded atom models often used for metals which incorporate the local environment isotropically, to complex models like the reactive empirical bond-order potential[90] developed to accurately model the chemistry of carbon

based materials and the reaxFF force fields that have been derived for a large variety of systems.[91,115] Molecular topology force fields are force fields for which specific interactions between atoms are differentiated. Examples include the CHARMM,[116] AMBER,[117] and OPLS[118] force fields available with many computational software packages. Defined interactions include bond-stretching, angle-bending, and torsional interactions, though other constrains can be applied to maintain less regular geometries. Interactions not explicitly defined are typically treated with Lennard-Jones potentials and Coulombic interactions that are applied to atomic pairs that lack specifically defined interactions. Employing these force fields requires more explicit definition of the initial condition, defining very specifically the interactions between atoms in the simulation model. Furthermore, it is important that the chemistry and behavior of the system be similar to the models against which the force field was parameterized, and typically the system should not be expected to undergo any major chemistry like bond breaking or forming, because changes in molecular topology are not normally incorporated in the parameterization scheme and add substantial complexity.

Two primary force fields are employed in this work, a two-body interaction potential used to generate silica substrates and a molecular force field used to model alkylsilane functionalized silica substrates. Specifically, the CHIK force field[119] was used to model the interactions between silicon and oxygen atoms in order to form the vitreous silica substrates onto which the molecules were eventually attached.. The interactions are defined as a combination of Coulombic interactions between the positively (negatively) charged silicon (oxygen) atoms, and a Buckingham potential interaction. This latter

potential is similar to a 12-6 Lennard-Jones potential typically used to describe long range interactions, but the repulsive interaction is instead represented as an exponential function. The total interaction potential is therefore defined as:

$$U(r) = A_{\alpha\beta} \exp\left[- r / \rho_{\alpha\beta} \right] - \frac{B_{\alpha\beta}}{r^6} + C \frac{q_\alpha q_\beta e^2}{r} \quad \text{..... (2.7)}$$

Wherein $A_{\alpha\beta}$ and $\rho_{\alpha\beta}$ represents the strength and range of the short range repulsive interactions, $B_{\alpha\beta}$ the strength of the long range attractive interactions, and C is a proportionality constant. The force field parameters are summarized in Table 2.1, and the potentials and force fields are depicted in Figure 2.1.

Table 2.1. Parameters for the CHIK force field employed for generation of the silica substrate according to Eq. 2.7.

Parameter	**Value**	**Parameter**	**Value**
$A_{O\text{-}O}$ (eV)	659.595398	$\rho_{O\text{-}O}$ (Å)	0.386091
$A_{Si\text{-}O}$ (eV)	27029.419922	$\rho_{O\text{-}Si}$ (Å)	0.193851
$A_{Si\text{-}Si}$ (eV)	3150.462646	$\rho_{Si\text{-}Si}$ (Å)	0.350699
$B_{O\text{-}O}$ (eV·Å^6)	26.836679	q_{Si} (e^- C)	1.910418
$B_{Si\text{-}O}$ (eV·Å^6)	148.099091	q_O (e^- C)	-0.955209
$B_{Si\text{-}Si}$ (eV·Å^6)	626.751953	C (eV·m/C^2)	14.399645

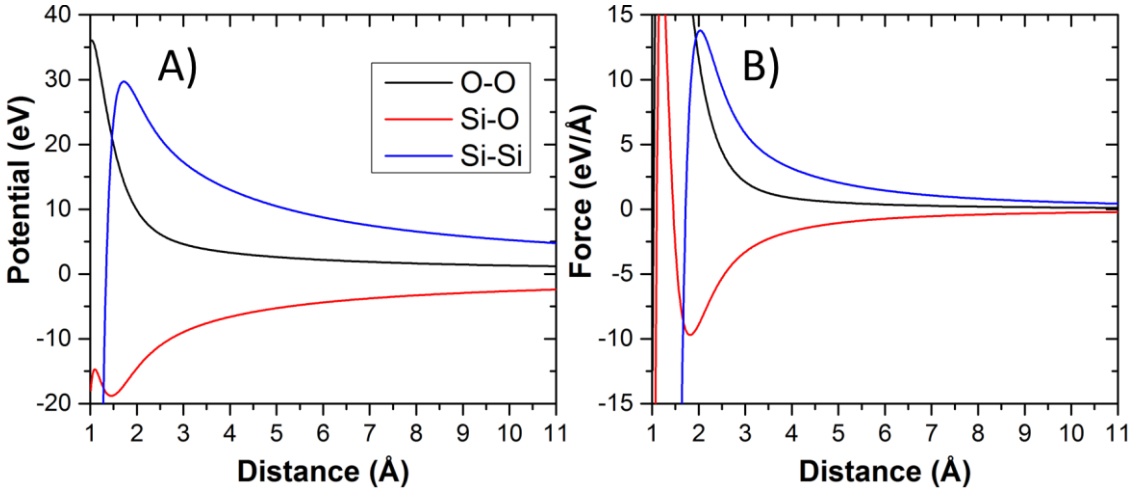


Figure 2.1. CHIK potential (A) and force field (B) for silicon and oxygen atom interactions. While the force field diverges at very low separations, as long as the temperature is sufficiently low, and the time step sufficiently small, atoms do not come sufficiently close for this interaction to dominate. O-O and Si-Si interactions are repulsive at all reasonable distances, and Si-O interactions bare a minimum at 1.5 Å.

The OPLS force field[118] was applied to the alkylsilane molecules.. Because the assembly of the film is driven by interactions between the molecules, this force field was chosen because it was parameterized against properties of organic liquids that also strongly depend on these interactions. In this force field, bonded and angle-bending interactions are treated as harmonic springs, torsional interactions are represented as a Fourier series in the torsional angle matching higher level calculations, and long range interactions between atoms are defined in terms of both Coulombic and a Lennard-Jones potential which captures Pauli repulsion at close range, and van der Waals attraction at long range. The equations, including figures depicting the specific interactions, are presented in Table 2.2, and the parameters employed in this work are presented in Table 2.3.

Table 2.2. OPLS Force Field description and interaction parameters.

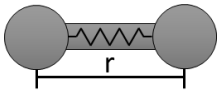
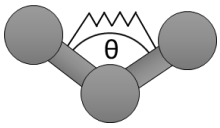
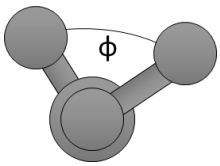
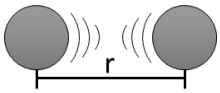
Interaction	**Equation**	**Parameter Definitions**
Bond-stretching 	$U_{bond}(r) = K_{ij}(r - r_{0,ij})^2$	K_{ij}: Bond strength (kCal/mol/Å) $r_{0,ij}$: Equilibrium bond distance (Å)
Angle-bending 	$U_{angle}(\theta) = K_{ijk}(\theta - \theta_{0,ij})^2$	K_{ijk}: Angle bending strength (kCal/mol/°) $\theta_{0,ij}$: Equilibrium bond angle (°)
Torsions 	$U_{tors}(\phi) = \frac{1}{2}K_{1,ijkl}(1+\cos(\phi)) + \frac{1}{2}K_{2,ijkl}(1-\cos(2\phi)) + \frac{1}{2}K_{3,ijkl}(1+\cos(3\phi))$	$K_{\alpha,ijkl}$: Fourier coefficient (kCal/mol)
Long-range 	$U_{pair}(r) = U_{coul} + U_{vdw}$ $U_{coul}(r) = \frac{C}{\varepsilon}\frac{q_i q_j}{r}$ $U_{vdw}(r) = 4V_{ij}\left(\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right)$	C: Proportionality constant 332.219 Kcal/mol Å/(unit charge)^2 ε: Dialectric constant 1.0 (vacuum) V_{ij}: Interaction strength (kCal/mol) σ_{ij}: van der Waals radius (Å)

Table 2.3. OPLS force field parameters employed in functionalized nanoparticle simulations. *Parameters for dissilimar interactions are taken as the geometric mean of their individual parameters. Long range interactions are ignored for atoms separated by one or two chemical bonds and attenuated by half for atoms separated by 3 bonds.

Long Range Interaction Parameters*			
Interaction	**V_{ii}(kCal/mol)**	**σ_{ii}(Å)**	**q(e^-)**
Si-Si	0.10	4.0	0.86
O-O	0.17	3.0	-0.43
OH-OH	0.00	0.00	0.418 (H), -0.683(O)
C-C	0.066	3.5	-0.12(-**C**H_2-), -0.18(-**C**H_3)
CH-CH	0.03	2.5	0.06

Bonding Parameters			**Angle Bending Parameters**		
Interaction	**K_{ij}(kCal/mol)**	**$r_{0,ij}$(Å)**	**Interaction**	**K_{ijk}(kCal/mol)**	**$\theta_{0,ijk}$(Å)**
Si-O	300	1.65	Si-O-Si	20	145
O-H	553	0.945	O-Si-O	60	110
Si-C	200	1.85	Si-O-H	23.78	122.9
C-C	268	1.529	O-Si-C	60	100
C-H	340	1.09	Si/C-C-H	37.5	110.7
			Si/C-C-C	58.35	112.7
			H-C-H	33	107.8

Torsional Parameters			
Interaction	**$K_{1,ijkl}$(Kcal/mol)**	**$K_{2,ijkl}$(Kcal/mol)**	**$K_{3,ijkl}$(Kcal/mol)**
Si/C-C-C-C	1.74	-0.157	0.279
Si/C-C-C-H	0	0	0.366
H-C-C-H	0	0	0.318

2.2 Generation of Alkylsilane Functionalized Surface Models

Preparation of the functionalized nanoparticles described in this work proceeded by several steps, the starting points consisting of a crystalline quartz structure which was annealed to generate vitreous silica and initially all-trans alkylsilane molecules defined as a silicon atom connected to an all-trans hydrocarbon chain; these initial structures are shown in Figure 2.2.

2.2.1 Generation of Vitreous Silica Nanoparticle and Flat Surface Substrates

The amorphous silica structure of the nanoparticles was generated by annealing the α-quartz starting material far above its melting point and then cooling the system rapidly. This produces an amorphous bulk structure from which the various substrates employed in the work can be produced. The CHIK force field[119] was employed for the vitrification

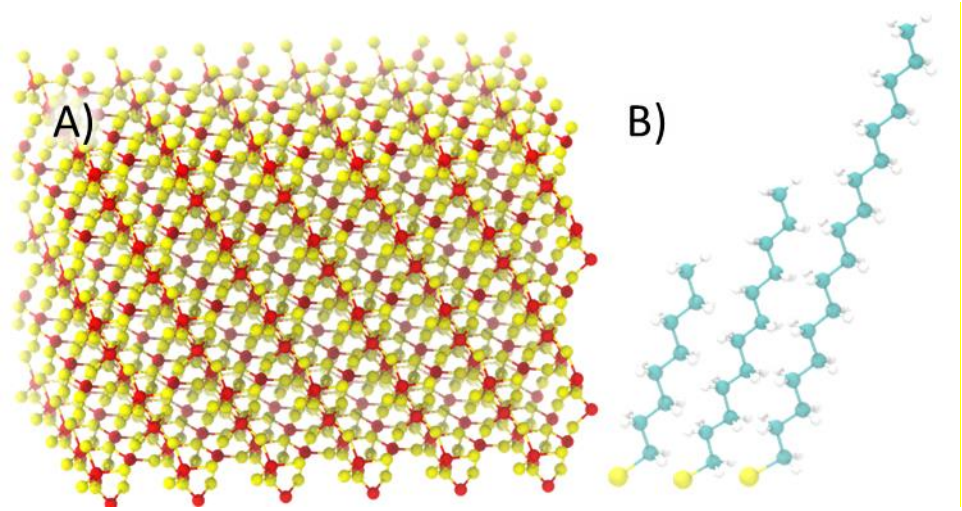


Figure 2.2. Initial structures used for building alkylsilane functionalized silica substrates, including an α-quartz structure (A) and all-trans octyl-, dodecyl-, and octadecylsilane (B), from shortest to longest.

step. This force field was recently developed as an improvement to the long-standing and often employed BKS potential,[120] which was itself parameterized against Hartree-Fock calculations of SiO_4. Several parameters were considered in the optimization of the CHIK potential, including temperature dependence of the bulk density, radial pair and angular distributions as compared to *ab initio* calculations, and vibrational density of states computations. As these parameters are fundamentally linked to the mechanical properties of silica, this force field was deemed ideal for generating the amorphous silica structures.

To generate vitreous silica, first, the initial α-quartz unit cell was replicated to generate a cell with dimensions 22.6×31.4×28.8 nm containing 1350 atoms. This was treated as an infinite bulk structure with periodic boundary conditions, and was annealed following the temperature profile depicted in Figure 2.3, wherein it was initially brought to 5000 K, cooled to 3600 K to compare its properties to the original formulation of the potential,

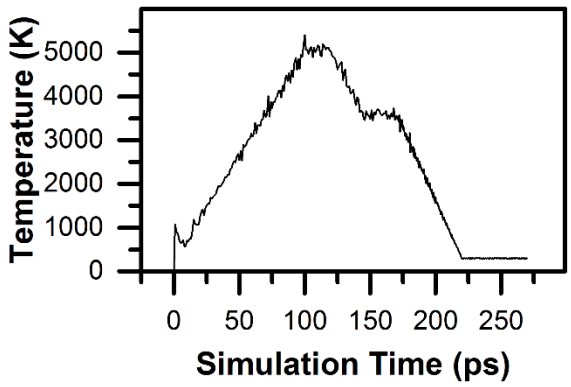


Figure 2.3. Annealing profile used to generate vitreous silica from α-quartz, the particle is first heated to 5000 K, briefly equilibrated, cooled to 3600 K and equilibrated for analysis, and finally cooled to 300 K to produce the final structure.

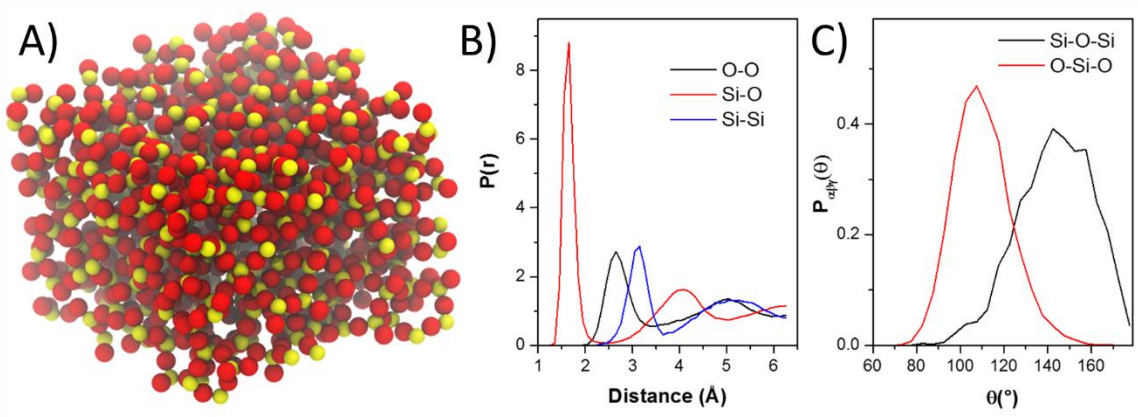


Figure 2.4. The unit cell of the final vitreous silica structure (A), as well as the radial pair distribution functions (B) and angular distribution functions (C) collected at 3600 K.

and finally cooled to 300 K to generate the substrates used in the work. The radial and distribution functions calculated at 3600 K are shown in Figure 2.4, along with the final angular vitreous silica structure, wherein excellent agreement with the original force field formulation was observed.[119]

Substrates were prepared by cleaving the infinite bulk vitreous silica structure. The structure was replicated over the periodic boundaries to generate a sufficiently large bulk structure from which to cleave the spherical and flat substrates. To generate particle surfaces, a spherical cleaving surface was employed, and for flat surfaces, periodicity in the z-direction was removed such that the cleaving surface was effectively the box boundary. All silicon atoms outside the cleaving surface were removed and all oxygen atoms greater than 2 Å from the outside of the cleaving surface were also removed, leaving an oxygen rich surface. This ensured that all silica atoms were fully coordinated and surface manipulation could be directed at the dangling oxygen atoms. For spherical

substrates, the inner core particles were removed to leave a 15 Å thick silica shell to minimize storage and processing requirements as the analyses in this work are directed at the surface behavior. Atoms within 3 Å of the back side of the surface structures were designated as fixed atoms to maintain the geometry of the substrate. Finally, to facilitate surface processing of the particles, atoms within 4 Å of the surface were designated as surface atoms, and the remainder of the atoms were designated as bulk, and bonds were assigned between all atoms using a cutoff of 1.8 Å. Figure 2.5 depicts the substrate preparation process for the 3.5 nm particle substrate.

The final processing step of the particle and flat surface structures involves hydroxylation of the surfaces and full conversion to a model description suitable for the

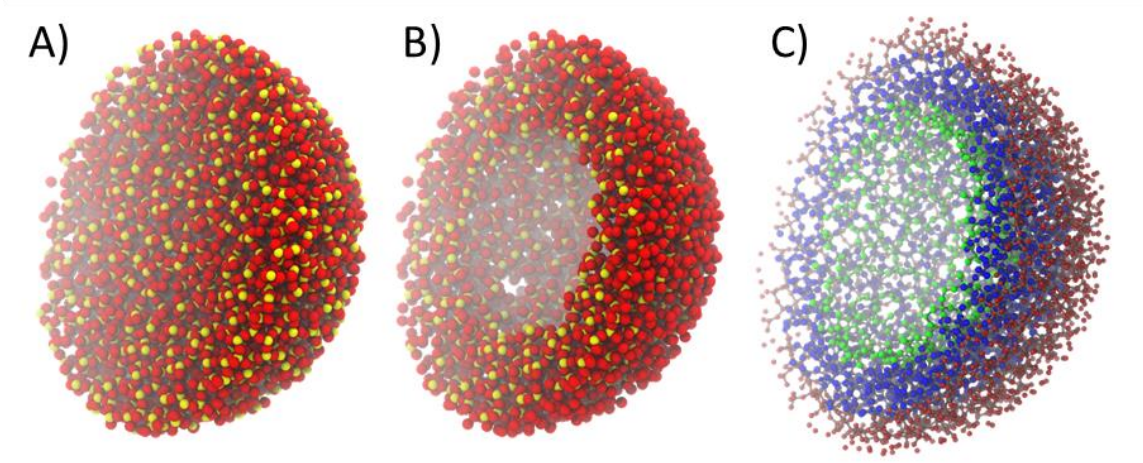


Figure 2.5. Depiction of particle preparation procedure, where cross sections of the particles are shown. First, the particle is cleaved from the bulk silica with an additional 2 Å of atoms (A), then the inner core is removed and all silicon atoms outside the specified radius are removed, leaving an oxygen rich surface (B), finally bonds are assigned and the core (green), bulk (blue), and surface (red) atoms are designated (C).

OPLS force field, requiring identification of angle-bending interactions and reassignment of atomic charges. To hydroxylate the surface, defect oxygen atoms were identified as those with only a single bond to silicon atoms, while oxygen atoms with no bonds were discarded. The defects were then hydrogenated by appending a hydrogen atom 1 Å from the defect oxygen, normal to the particle surface. A final post-processing step was conducted to ensure that the hydroxyl defect densities were realistic, reflecting appropriate density of –Si(OH), -$Si(OH)_2$, and –$Si(OH)_3$ groups.[121] To achieve this, silanol groups with multiple hydroxyl groups were condensed based on proximity, with the proximity cutoff varied for each particle surface to produce the expected concentrations. Densities of the resulting silanol groups and the target concentrations are reported in Table 2.4.

The hydroxylated silica nanoparticle was then relaxed. In the first step, a cosine potential was employed for pairwise interactions to gently separate any close contacts that would have been unstable under the Lennard-Jones potential employed by the OPLS force

Table 2.4. Densities of silanol groups on the silica nanoparticle surfaces compared to reported densities.

	Expected silanol density[121]	7 nm		12 nm		40 nm		Flat	
		Count	%	Count	%	Count	%	Count	%
-Si(OH)	83%	462	81%	1333	81%	14431	81%	952	80%
-$Si(OH)_2$	17%	108	19%	309	19%	3429	19%	240	20%
-$Si(OH)_3$	0%	0	0%	0	0%	0	0%	0	0%

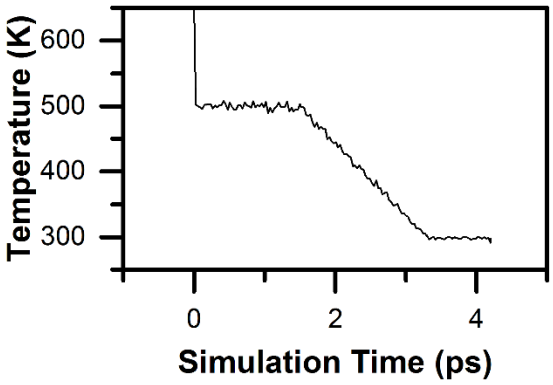


Figure 2.6. Temperature profile for final equilibration of the hydroxylated silica substrates. A brief, 500 K anneal cycle was initially performed and then the substrate was cooled to 300 K and equilibrated.

field. Then, under the full OPLS potential, the model was relaxed according to the temperature profile depicted in Figure 2.6. Final, equilibrated, hydroxylated nanoparticles and flat substrates are depicted in Figure 2.7.

2.2.2 Functionalization of Silica Surfaces

The final step in generating the simulation models involved the addition of the alkylsilane film. A uniform distribution of the film molecules was chosen, this was achieved by positioning each additional molecule so as to maximize the distance from all previously attached molecules. The precursor molecules, depicted in Figure 2.2, were constructed such that the hydrocarbon backbone was directed along the x-axis, and the silicon atom to be attached to the substrate was placed at the origin, and the particles themselves were positioned so that their center lied at the origin. The molecules were attached to hydroxyl groups, such that the silicon atom of the alkylsilane assumed the position of the original hydrogen atom, and the hydrocarbon chain was projected normal

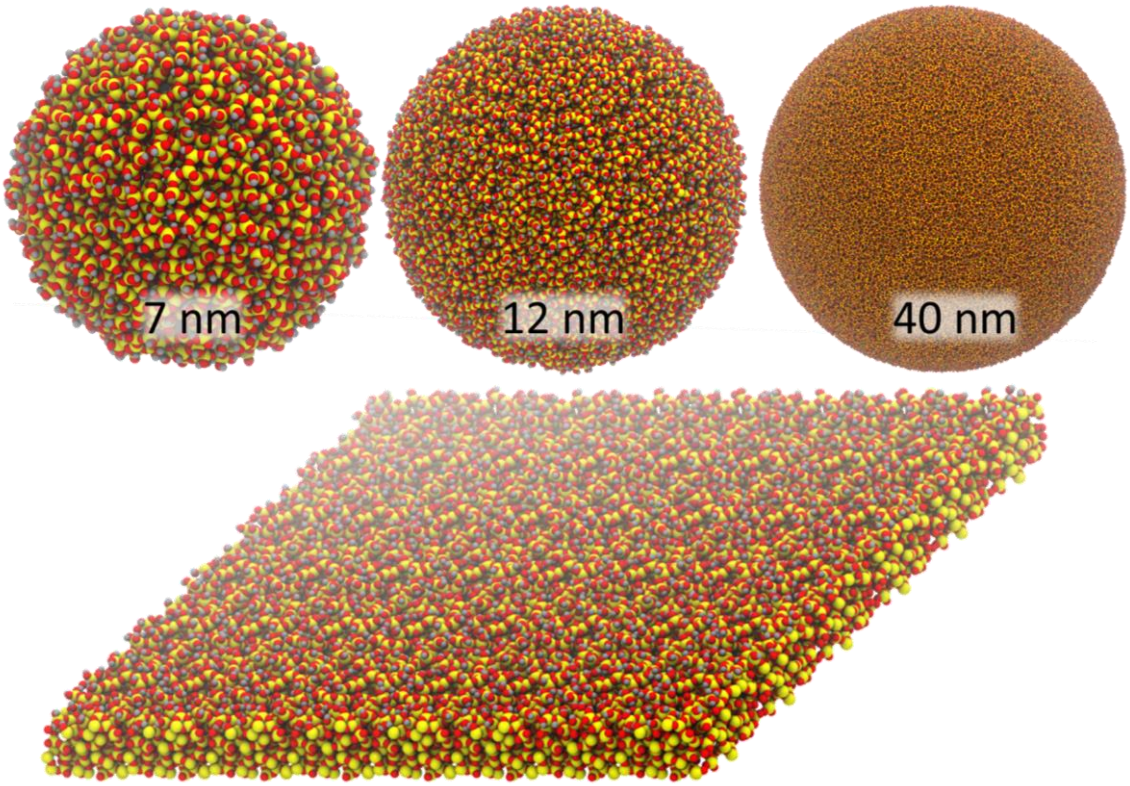


Figure 2.7. Final hydroxylated silica substrates of the various diameters considered, as well as the flat surface, prior to functionalization.

to the surface. This was achieved by generating a unitary rotation matrix which rotates the x-axis to the surface normal and the y- and z- axes to two perpendicular surface tangents, this is depicted in Figure 2.8. Generation of the unitary rotation matrix is achieved by the following equations:

$$\begin{matrix} \vec{v}\cdot\vec{d}=0 \\ \vec{v}\times\vec{d}=\vec{u} \end{matrix} \qquad M=\left(\vec{v}^{tr} \quad \vec{d}^{tr} \quad \vec{u}^{tr}\right) \qquad \text{..... (2.8)}$$

Wherein $\vec{v}$ is the direction vector from the particle center to the attachment site, $\vec{d}$ is a vector chosen so that it is perpendicular to $\vec{v}$, and thereby tangent to the surface, and $\vec{u}$, a

vector perpendicular to both $\vec{v}$ and $\vec{d}$, and also tangent to the particle surface. Composing a matrix of these normalized direction vectors yields a unitary transformation matrix used to rotate the molecule from alignment on the x-axis, to alignment with the surface normal. For functionalization of flat surfaces, the molecules were rotated to align with the z-axis, corresponding to the surface normal, and similarly placed so that the silicon atom was positioned where the hydrogen atom of the hydroxyl group originally resided. Images of the functionalized particles of radius 3.5 nm are depicted in Figure 2.9, and a complete set of images of the functionalized particles is presented in Appendix A. Additionally, the software used to construct and subsequently analyze the functionalized particles is presented in Appendix B.

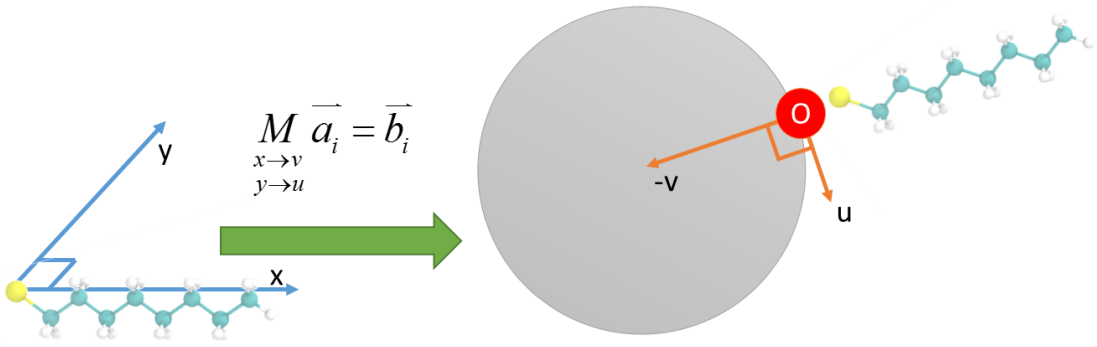


Figure 2.8. Attachment scheme for placing molecules on the surface, by which a unitary rotation matrix U, that takes the molecular axis vectors x and y to the surface normal and tangent respectively, is applied to all molecular particle positions a_i to produce functionalized particle positions b_i.

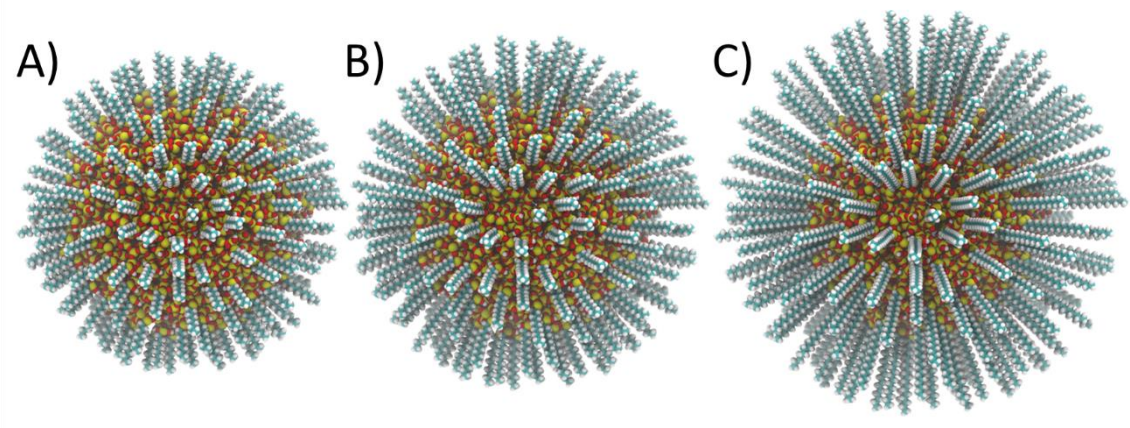


Figure 2.9. Nanoparticles of radius 3.5 nm immediately after functionalization with octyl- (A), dodecyl- (B), and octadecylsilane (C) with surface coverage of 1.5 molecules/nm^2.

2.3 Scanning Probe Microscopy and Lithography Techniques

2.3.1 Scanning Tunneling Microscopy

STM is an effective technique for imaging surfaces in three dimensions with atomic resolution. It was developed 1984 by Binhig and Rohrer,[122] and has been the basis for several off-shoot techniques like AFM[123] and SNOM.[124] In this technique, as with all SPM techniques, a probe is scanned across the surface, and a feedback loop maintains the vertical position of the probe with respect to the surface in response to a signal, and by mapping the vertical position of the tip versus the lateral position of the probe, a topographic map of the surface can be generated. In STM, the signal that drives the feedback loop is the tunneling current between the tip and the substrate. A tunnel current is effectively the flow of electrons through a classically forbidden region, an effect that is only observable for gaps on the order a several Å to a few nanometers, and which is highly sensitive to the gap width.

The excellent resolution of the STM arises from two key factors: control of the tip position with piezoelectric materials and the exponential relationship between the current flow and the tip-sample separation in a tunnel junction. Piezoelectric materials are voltage responsive ceramic materials that expand and contract under an applied bias, and their resolution depends on the size of the piezoceramic and the precision with which the bias is applied. High resolution instruments can achieve resolution on the order of 0.1 Å typically. The exponential dependence on gap width arises from the fact that the tunnel junction acts as a potential barrier to the flow of charge carriers. Electrons with energy lower than the potential barrier must tunnel through the gap, and the rate at which this occurs depends on the penetration of the carrier wave function through the tunnel gap. This is depicted in Figure 2.10. In addition to the enhanced sensitivity to the gap width, a key result of this dependence is that the vast majority of current flow occurs through the apex of the STM tip. If a single tip atom is closest to the surface, it will be responsible for imaging the surface, such that the lateral resolution is dictated by the size of the apex atom, on the order of 1-2 Å.

The detection signal in an SPM technique largely dictates the source of contrast when imaging. In STM the signal is a tunneling current, so contrast arises from both the local geometric structure and electronic properties of the surface. Specifically, the tunnel current is proportional to the overlap of the electronic states of the tip and the surface:

$$I \propto \left| \langle \Psi_t | H | \Psi_s \rangle \right|^2 \quad \text{..... (2.9)}$$

The overlapping states that contribute to conduction are those states with energies between the chemical potential of the tip and the surface, and contrast therefore depends on the

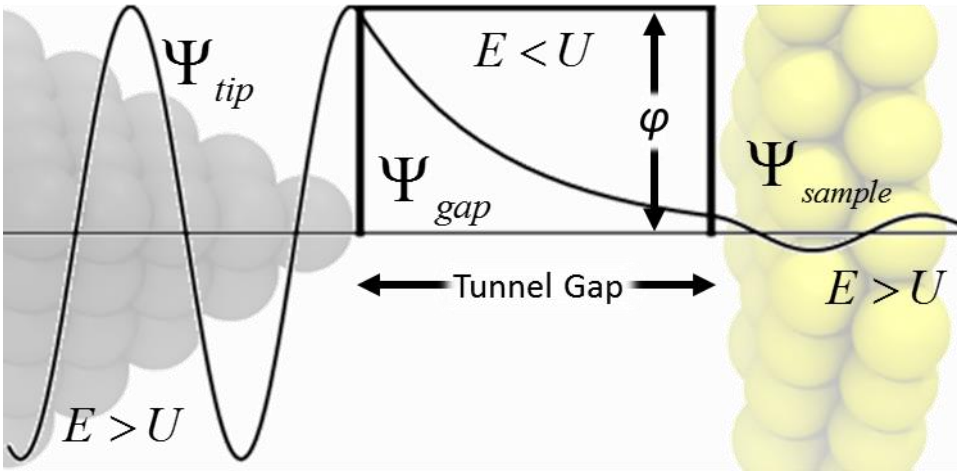


Figure 2.10. Schematic of an STM tunnel junction, the potential barrier, and the electronic wave function, demonstrating the basis for the exponential dependence on tunnel current with gap width. Reproduced with permission from Ewers, B. W.; Schuckman, A. E.; Batteas, J. D. "Why Did the Electron Cross the Road? A Scanning Tunneling Microscopy (STM) Study of Molecular Conductance for the Physical Chemistry Lab". *J. Chem. Educ*. **2014**, 91, 283-290.[125] Copyright 2014 American Chemical Society.

direction of current flow and the magnitude of the bias applied between the tip and the surface. For example, when imaging a surface, a strong positive tip bias will allow for imaging of the unoccupied states of the surface, and a strong negative tip bias will image the occupied band states of the surface, allowing for specific imaging of molecular HOMO and LUMO orbitals[126,127] or different states of semiconductor surfaces.[128,129] Similarly, the topographic contrast can be used to examine the local conductivity of the surface that arises due to defects and adsorbed molecules. If the geometric structure is known, then the electronic contributions can be isolated, providing an effective means for characterizing the electronic properties of molecules solely from STM topographic information.

STS represents a variation in STM wherein the lateral tip position is held fixed, and the applied bias or tip-surface separation, or both, are varied. By varying the tip bias, the available states by which conduction may occur is varied,[130] allowing for more detailed analysis of the local electronic structure of a surface. By varying the tip position, the properties of the tunnel gap can be investigated. For example, the tunneling efficiency of the gap, typically denoted β in the equation:

$$I \propto e^{-\beta z} \quad \text{.....} \quad (2.10)$$

This has, for example, been employed to examine the tunneling efficiency of different components of molecules adsorbed to surfaces.[131] These spectroscopic techniques complement STM imaging, providing more extensive information on the dependence of current flow in terms of the applied bias and tip position.

2.3.2 Atomic Force Microscopy

AFM was developed shortly following the development of STM.[123] Like STM, the tip position is controlled by piezoceramic materials, providing exceptional lateral and vertical resolution of the surface, and a feedback loop is used to control the vertical position of the probe. In traditional, contact-mode AFM, the signal governing the action of the feedback loop is the force applied to the surface, rendering the AFM sensitive to the mechanical properties of the surface. As such, the surface does not need to be conductive, as is the case in STM, but it must be sufficiently mechanically rigid to support the pressure of the AFM tip and the shear forces as the tip images the surface. Modulation techniques like tapping mode and non-contact mode AFM are even capable of imaging soft surfaces with complete removal of any shear stress on the surface, though they differ in the

information provided about the surface, resolution, and accessibility.

Similar to STM, in contact-mode AFM, a topography image is generated by mapping the vertical position of the probe that is controlled by the feedback loop, as a function of the probe's lateral position. The probe is a nanoscopically sharp tip suspended from a cantilever, typically fabricated from silicon or silicon nitride. By bouncing a laser off the back of this cantilever onto a four-quadrant photodiode, the vertical deflection of the laser can be measured, which for small deflections is proportional to the force applied to the surface. The laser deflection signal is therefore used to drive the feedback loop in order to maintain a constant force on the surface. Contrasting with STM, the vertical resolution is nominally lower because the laser deflection is linearly, not exponentially, related to the vertical position of the AFM tip. Additionally, lateral resolution is typically lower because it is dictated by the size of the contact between the AFM tip and the surface. The AFM tip cannot be expected to be atomically sharp as such a sharp asperity would be incapable of supporting the load on the AFM tip, and rather the contact area is at least several Ångstroms wide. Lattice resolution has been observed, but this is attributed to stick-slip friction between the AFM tip and the surface, not the surface topography, and single atom defects that would be visible in STM cannot be observed in AFM except in special cases. Non-contact mode AFM, a technique in which the signal used to drive the feedback loop is the phase shift or change in amplitude of a vertically oscillating tip, provides lateral resolution down to the atomic and molecular scale, but this technique is generally not as versatile and is more difficult to implement then contact-mode AFM.

Characterizing the AFM Probe

Quantitative control of the interactions between the AFM probe and the surface requires careful characterization of both the cantilever stiffness, which dictates the forces exerted, and the tip geometry, which dictates the interaction area. Together, these two factors ultimately govern the distribution of pressure exerted upon the surface. To determine the cantilever stiffness, preliminary information about the cantilever dimensions are required. The general formula for the dynamic spring constant, which is proportional to the cantilever stiffness, is:[132]

$$k_d = \rho b^2 L \Lambda\left(\mathrm{Re}\right) \omega_R{}^2 Q \quad \text{..... (2.11)}$$

Where ρ is the fluid density of the medium, b is the cantilever width, L is the cantilever length, ω_R is the resonant frequency of the cantilever, and Q is the quality factor, which is an expression of the rate of energy dissipation when the cantilever oscillates at its resonant frequency. The function Λ(Re) reflects the geometry of the cantilever, and has been empirically determined for a variety of geometries including the rectangular and triangular geometry cantilevers employed in this work.[132] Note that the general equation does not depend on the tip material or thickness. To use this equation, it is necessary to know the dimensions of the cantilever, which are typically provided by the manufacturer or can be measured directly. The resonant frequency and quality factor must be measured for each specific cantilever, as the actual values for a given cantilever often deviate quite widely from the values reported, and the medium in which the resonant frequency and quality factor are determined, typically air, dictates the fluid density (air = 1.18 kg/m^3).

In order to determine the resonant frequency of cantilevers used in this work, the power

spectrum of the cantilevers was measured. This is achieved using a small piezo driver coupled to the AFM probe, driven over a range of frequencies. Using lock-in amplification, the amplitude of the tip vibration at these frequencies was measured. Determination of the resonant frequency and quality factor was achieved by fitting the resulting power spectrum to the harmonic oscillator function:[133]

$$P(\omega) = \frac{0.001 H \omega_f{}^4}{\left(\omega^2 - \omega_f{}^2\right) + \omega^2 \omega_f{}^2 / \mathrm{Q}^2} \quad \text{..... (2.12)}$$

Where H represents the amplitude of the resonant peak, ω_f the center of the peak, and Q the quality factor. An example of a resonance curve and fitted function is depicted in Figure 2.11. Combining this information with the properties of the tip, the stiffness of the AFM cantilever was determined.

The total sensitivity of the AFM, i.e. the relationship between the amount of laser

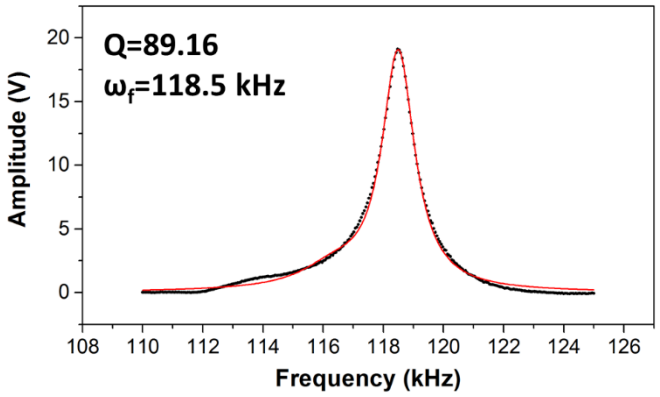


Figure 2.11. A measured resonance curve (black) and fitted harmonic oscillator function (red) used to determine the quality factor and resonant frequency of an AFM cantilever.

deflection and the force applied to the surface, requires one remaining factor, the deflection sensitivity. This can be determined by measuring the laser deflection as a function of the probe's vertical position. Once the tip is in contact with the surface, for small deflections, the slope of the curve corresponds to the deflection sensitivity. The total sensitivity is thus determined by the equation:

$$C\left(nN/\mathrm{V}\right) = 1000 \times {}^{k(N/m)}\!\Big/_{FD(V/\mu m)} \quad \text{..... (2.13)}$$

Where k is the cantilever stiffness and FD is the deflection sensitivity, determined in the units indicated. These two key pieces of information are therefore essential for quantitative control of the force applied to the surface.

In order to control the pressure applied to the surface, the tip geometry must be known. A straightforward method of determining the tip geometry is through reverse imaging, that is, a surface with known structure is imaged, and a resulting image of the tip can be generated through deconvolution. To image the tip apex, the surface features must be sharper than the tip, and atomically sharp structures are therefore ideal. The (305) surface of strontium titanate ($SrTiO_3$) is an excellent candidate, this hard material, when annealed in oxygen, produces atomically sharp ridges that can be imaged, providing a profile of the AFM tip.[134] An AFM image of the $SrTiO_3$ surface is shown in Figure 2.12, where the rounding of the atomically sharp edges is a result of the curvature of the tip. By rotating the sample, different profiles of the AFM tip can be determined. While calibration surfaces exist that can provide full, two dimensional imaging of the tip apex, $SrTiO_3$ represents the best case, consistently atomically sharp structure. Furthermore, when applying the nanografting technique discussed in the next section, it is most important to

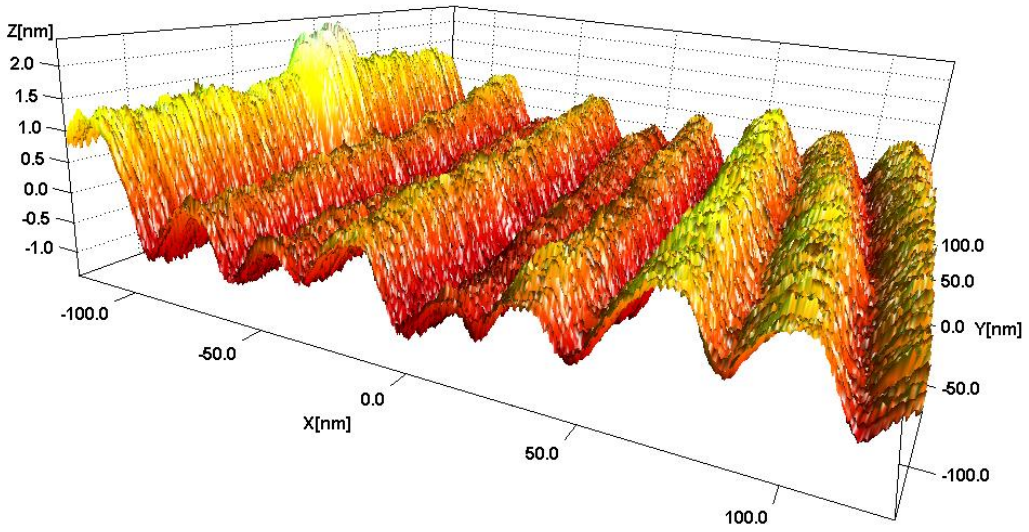


Figure 2.12. Image of the $SrTiO_3$ substrate used to determine the radius of curvature of an AFM tip. The atomically sharp ridges of the substrate appear rounded due to the tip geometry, and from this, the tip shape can be deconvoluted.

know the profile of the AFM tip perpendicular to the sliding direction, as this dictates the line width, and therefore one dimensional profiling is sufficient.

AFM Nanografting

A variety of AFM-based nanolithography methods have developed in recent years.[135] The most prevalent due to its simplicity is dip-pen nanolithography, virtually a nanoscale version of writing with pen and paper. These techniques take advantage of the high resolution afforded by the AFM in addition to the fact that the AFM probe directly interacts with the imaged surface. In the nanografting surface patterning approach used herein,[136] a surface is functionalized with a SAM, and the AFM tip is used to displace this

matrix SAM from the surface. A wide range of SAMs and surfaces can be used,[137,138] however thiols on Au surfaces are the most commonly employed. When this is done in a clean solvent, so-called nanoshaving produces an unfunctionalized patch on the surface. If this is done with another molecule in the solution which can bind to the surface, the other molecule will back-fill the freshly exposed surface, this is depicted in Figure 2.13. Importantly, while alkanethiol SAMs typically require several hours to achieve self-assembly, nanografted features are observed to assemble immediately, suggesting that nanoconfinement of the adsorbing molecules between the tip and the surrounding matrix facilitates assembly of these molecules as they adsorb to the exposed surface.[139] Furthermore, by confining the adsorbed molecules in the matrix SAM, molecules that

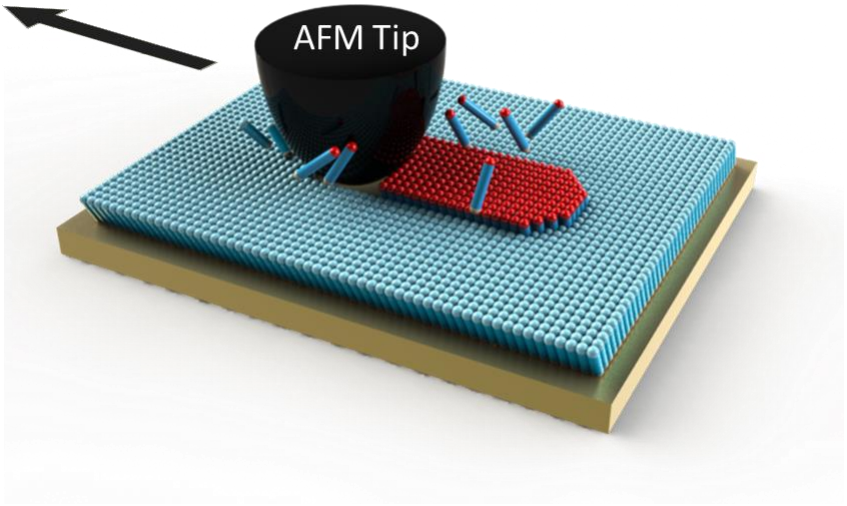


Figure 2.13. A depiction of the nanografting process, wherein a SAM matrix (blue) is shaved away by an AFM tip, while a target molecule (red) adsorbs to the freshly exposed surface.

cannot effectively assemble on their own can be driven to assemble via this nanoconfinement effect.

In this work, control of tip motion during nanografting was achieved by a scripting interface detailed in Appendix D. The benefits of using a custom scripting interface, instead of the imaging interface available with the software, are primarily technical, with the scripting interface providing higher throughput and more direct control over the tip position. Two-dimensional box features were fabricated by rastering the tip back and forth across the surface with specified line spacing. In all cases, the grafts were patterned such that the fast axis, that is the direction the individual lines were patterned, was aligned with the cantilever axis, so that torsion of the cantilever does not impact the grafts. This is particularly important for the smallest grafts, where the restoring force applied to the tip apex may not be sufficient to overcome the static friction force holding the probe in place. The cantilever is more rigid along the cantilever axis, providing a stronger and more responsive restoring force, ensuring that the tip moves in response to the piezo translations applied.

Pattern Relocation Scheme

A key goal of this work is to examine how fabricated structure size influences electronic transport properties, but the AFM used to fabricate the structures is incapable of interrogating their electronic properties. While CP-AFM could potentially be employed, typical conductive probes have a metal coating which is soft and easily damaged, so the nanografting process would likely degrade the tip. It was therefore necessary not only to fabricate these structures, but to also be able to efficiently relocate

them with other scanning probe techniques like STM. Several drawbacks of the nanografting technique render relocation of patterned features a significant challenge. Without aide in the relocation process, the area of surface to be considered could be presumed to be roughly 1 mm^2. Assuming a patterning process with minimal contributions from setup and tip translations, where the tip speed is 100 nm/s and typical line widths are 1 nm, it would take about one week to modify just 0.01% percent of the area of the surface. This is completely impractical for a variety of reasons. To minimize degradation of both the SAM and the sanity of the user, a typical patterning period of 8 hours is typical, and in this timeframe about 0.0003% of the surface could be modified. In the relocation step, the size of the search scans must take into consideration the size of the fabricated features and the limits of the instrumentation. For large features, search scans of 10x10 μm are feasible with modern high resolution SPM instruments, however the target dimensions considered in this work are 5-50 nm, requiring search areas no greater than 1x1 μm, or ~0.0001% of the total search area. Taking both the rate of surface modification and surface searching into consideration, the pattern relocation process is effectively insurmountable. This analysis, however, offers a prescription for how this challenge can be overcome. First, by reducing the total search area, these numbers become less challenging. Second, preparing patterns that are easy to identify in large searching scans improves the search process itself.

The first step in this scheme was to define the search area. Patterning in all cases was performed on Au(111) surfaces on mica, this soft surface is easily modified and scratched, and this can be done with considerable precision using AFM. A visible modification of

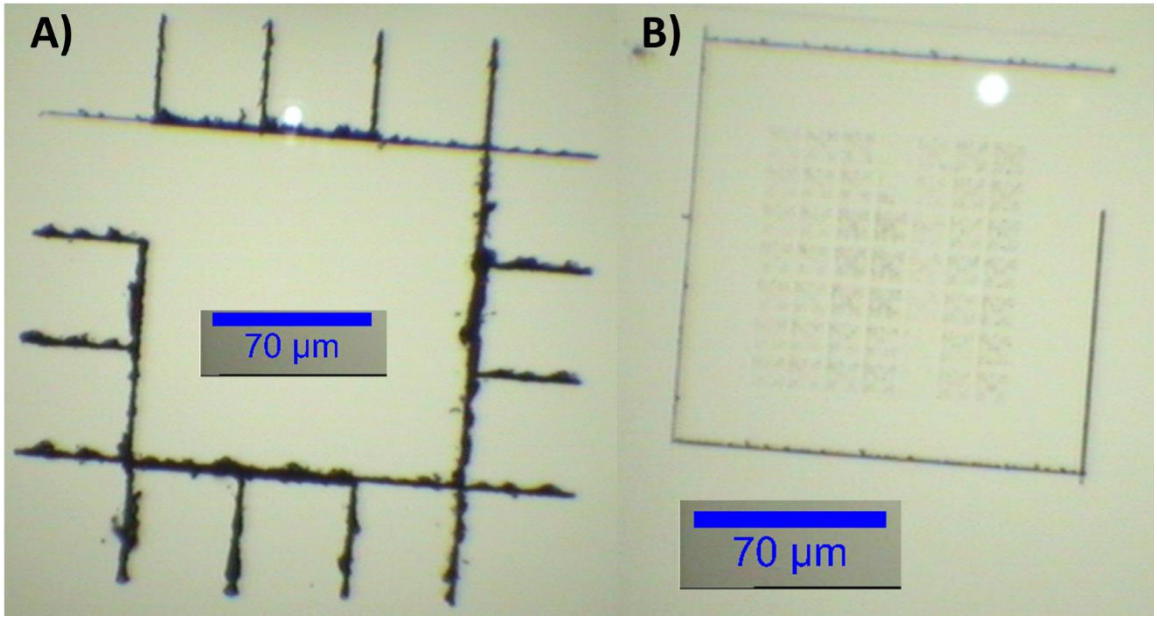


Figure 2.14. Examples of box confinement patterns. The tick marked box (A) is ideal for samples in which the subsequent star patterns are nanografted into the surface, providing a guide for tip placement during the patterning process. A simple box can be used when star relocation patterns (Figure 2.16) are themselves carved inside the box (B).

the surface was employed to restrict the search area. Using an extremely stiff AFM tip with an extremely high applied force, a box of 150x150 um was carved into the Au surface. These scratches are visible in the AFM employed for nanografting, and the STM where the features would be later investigated, so that probe placement within the search area was not a challenge. Examples of box patterns that were used are shown in Figure 2.14.

In order to facilitate larger searching scans, a star pattern was employed that could be easily observed in search scans several microns wide. The star patterns employed were 10 µm wide, with a 2 µm unmodified area in the center where the nanografting experiment was conducted. These sparse patterns could be quickly produced and offered a speedy

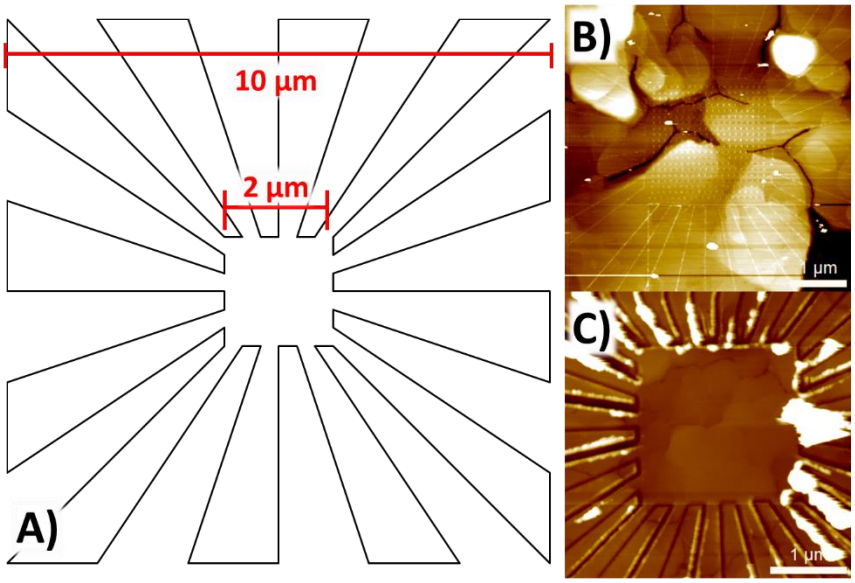


Figure 2.15. A schematic of the star patterns employed to facilitate pattern relocation (A) and examples in which the star is prepared by nanografting (A) and by carving the surface with an AFM tip (C).

means to modify a 10x10 µm area of the surface. The sides of the star consist of converging lines, so that once a star pattern is located, following the convergence of the lines the central test area could be found, and this star pattern is depicted in Figure 2.15A. These star patterns were made in two different ways, depending on the conditions of the grafting experiment. They were either fabricated during the nanografting experiments as nanografted features themselves, or they were scratched into the surface in a similar fashion as the box pattern, though with less force applied. Images of star patterns prepared using both approaches are presented in Figure 2.15B and 2.15C. During the search process, a nanografted star provided clear indication that nanografting was performed inside the star feature. For prefabricated star structures, an indexing scheme was

employed because not every star structure was expected to contain nanografted features. Before nanografting in a given star, it was identified by these indices, and this could be correlated in subsequent imaging experiments. Corresponding AFM and STM images of a box index is shown in Figure 2.16. Without these relocation mechanisms, pattern relocation was virtually impossible. By simply restricting the search area, relocation times were reduced to several days, and with addition of the star structures, relocation could be achieved within a few hours, allowing for quick analysis of nanografted features.

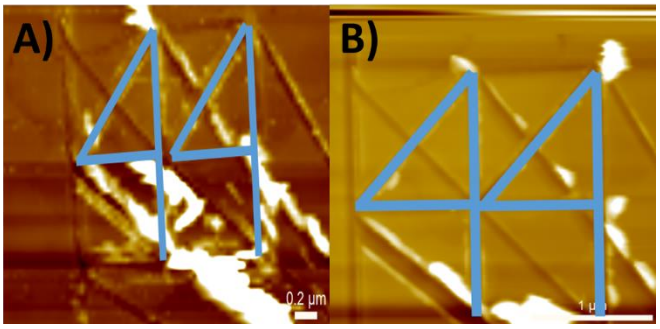


Figure 2.16. Indexing figures of carved star structures imaged by AFM (A) and STM (B), these indices could be used to correlate nanografting activities and further facilitate pattern relocation.

CHAPTER III

MOLECULAR DYNAMICS SIMULATIONS OF ALKYLSILANE MONOLAYERS ON SILICA NANOASPERITIES: IMPACT OF SURFACE CURVATURE ON MONOLAYER STRUCTURE AND PATHWAYS FOR ENERGY DISSIPATION IN TRIBOLOGICAL CONTACTS[*]

3.1 Introduction

Understanding the capabilities and limitations of SAMs as wear reducing films in nanotribological applications represents a critical step toward determining how best to achieve self-sustaining, robust lubrication systems necessary for the implementation of dynamic MEMS devices. One of the largest impediments to achieving this is the detrimental stiction and wear that occurs at the interfaces within such devices.[140,141] At the heart of the problem is the silica layer that naturally forms on the silicon based parts of MEMS devices, which presents an abundant number of high energy surface sites that are susceptible to interfacial tribochemistry,[142] including the adsorption of water which both catalyzes surface degradation and magnifies adhesive forces via capillary effects.[143] While alternative materials in device fabrication like DLC have shown promise,[144,145] it will be some time before these approaches are as capable and cost-effective as the more highly matured silicon microfabrication. Considerable research has focused on

[*] Reproduced with permission from Ewers, B. W.; Batteas, J. D. "Molecular Dynamics Simulations of Alkylsilane Monolayers on Silica Nanoasperities: Impact of Surface Curvature on Monolayer Structure and Pathways for Energy Dissipation in Tribological Contacts". *J. Phys. Chem. C*. **2012**, 116, 25165-25177. Copyright 2012 American Chemical Society.

developing lubricant systems that minimize interfacial forces and which are capable of surviving the intentional or intermittent contact found in MEMS devices.[146] The viscosity of traditional liquid lubricants is too high and prevents device motion, and the majority of lubrication schemes therefore involve the application of surface coatings to both reduce friction forces and wear. These schemes include the application of polymers,[147-150], ionic liquids, [151,152] diamond like carbon films,[153,154] layered inorganic films[155,156] such as MoS_2,[157-159] vapor phase lubricants,[25,160-162] and the use of monolayers derived from organosilanes.[36,140,141,163,164] Common to all of these is an attempt to reduce surface energies and shear forces that result in tribochemistry, surface deformation, third body formation, and ultimately destruction of the interfaces and device failure.

The use of molecular monolayers, SAMs, is the primary focus of this work, however recent developments suggest that the closely related approach of vapor phase lubrication will be a more successful means of achieving lubrication in MEMS devices.[25,160] Both approaches employ small organic molecules, with vapor phase lubrication achieving greater success likely because the lubricant molecules are constantly replenished from the gas phase, whereas a SAM will only reduce friction and prevent wear of the underlying surface until it is worn away, a seemingly unavoidable situation.[146,165] The primary drawback of vapor phase lubrication is the substantial support required to maintain the lubricant vapor without violating the low-profile of the MEMS device, one of its strongest benefits. Researchers at Sandia National Laboratories have developed polymer release systems[166] which could be incorporated directly into devices to solve this problem. Another approach combines the notion of a statically bound film and a mobile lubricant.

It has been found that alkylsilane SAMs infused with the small molecule,[167] 3P1P demonstrate considerably better wear resistance than the untreated SAM, and the focus of ongoing work is to understand how this composite system achieves wear reduction. This requires a better understanding of the structure of SAMs in devices and the mechanisms of their failure.

SAMs attracted the attention of the nanotribology community for a variety of reasons. Their application is straight forward, being prepared by either solution[31] or vapor phase[168,169] deposition. They are chemically robust within typical operating limits as they consist of simple aliphatics bound to the surface by strong covalent bonds. They lack conformational rigidity, allowing for energy dissipation through bond deformation pathways rather than destructive tribochemistry. Such pathways include compression of the film through the formation of *gauche* defects and local changes in the molecular tilt angles.[170-172] Tight packing within SAMs reinforces these deformation pathways which allows for greater energy dissipation without irreversible chemical changes. Though SAMs have been shown to successfully mitigate adhesive forces in devices,[36] and they demonstrate better frictional characteristics than bare silica surfaces,[170] their tendency to degrade on contacting and shearing interfaces[146,165] presents a significant obstacle. Nanoscale roughness of the surfaces in MEMS devices, though desirable to minimize adhesion,[173,174] plays a considerable role in their failure at interfaces,[175] and a goal of our ongoing work[33,176] is to understand how nanoscale roughness effects the quality and effectiveness of SAMs as lubricant films.

Surface roughness alters the nature of interfacial contacts,[5,177] and has two prominent

effects. First, and perhaps most critical, contact between surfaces occurs with the greatest pressure and shear at nanoscale[178] asperity-asperity contacts, therefore any attempt to alleviate wear in these systems must focus on these most extreme contact conditions. Second, and the primary focus of this work, SAM formation and structure are compromised by surface curvature due to greater free volume in the radially confined monolayer.

Developing a working understanding of the contact mechanics of rough surfaces in contact presents a considerable challenge. Accessing a solid-solid interface with experimental methods is a very difficult task with considerable limitations. Simulation methods on the other hand are generally inhibited by the many lengthscales required to study rough-on-rough interfaces. The most common approach to studying the contact of rough solids involves the application of continuum mechanics and finite elements methods. Mark Robbins and coworkers have led the way in modeling these contacts using finite elements methods,[179,180] but because these methods lack atomistic detail, there are difficulties both accurately modeling contact,[181,182] and understanding the chemistry of the solid-solid interfaces which is critical to understanding the effects and degradation mechanisms of lubricant films. Methods that combine atomistic and continuum mechanics models have been developed[183] which resolve many the issues in accurately modeling the contact mechanics of rough surfaces,[11,184] but they are yet to be applied to understand the effects, properties, and dynamics of adsorbed films.

Due to these difficulties, studies attempting to systematically investigate isolated single-[185-187] and double-asperity[33,176] contacts lubricated by SAMs have been employed

to simulate the asperity contacts at rough solid-solid interfaces. Indeed, AFM, a commonly used tool in the study of nanotribology of wear-reducing films, inherently simulates a single nanoscopic asperity contact, but without likewise enforcing roughness on the substrate on the same length scales, true asperity-asperity contacts are not accurately simulated. Surface curvature is observed to increase disorder within alkylsilane SAMs,[32,33] and characterizing these curvature effects and their overall impact on the structure and stability of molecular films provides an important route toward developing and improving these approaches for sustainable wear reduction. Thus, understanding these films in molecular detail and at interfaces that simulate the surfaces found in a MEMS device (*i.e.* on nanoscale asperities) is critical to guiding the improvement of these films. While the effects of surface curvature on SAMs have been studied on gold nanoparticles with thiol monolayers,[188-190] direct comparisons between thiol and silane SAMs must be made carefully due to the notable differences in the formation mechanisms of these films and the differences between the surface chemistries and molecular packing densities of these two systems.

Among the primary goals of applying SAMs to surfaces within MEMS devices are the reduction of surface energies and the maximization of routes of energy dissipation which do not lead to surface reactions, *i.e.* tribochemistry. The methyl termination of these SAMs presents much lower surface energy than the underlying silica, and energy dissipation during shear and impact may be achieved via reversible conformational deformations at a molecular level.[171,191] It is also beneficial to maximize the energy dissipated through conformational pathways such as *trans* to *gauche* bond rotations. Tight

packing in such films increases the energy required to undergo deformation, as clearly indicated by the relatively low (*ca*. 5%)[192] density of defects typical of films derived from octadecylsiloxanes. Such low naturally occurring defect densities imply greater deformation energy, allowing for greater non-destructive energy dissipation pathways within the film. Along with managing the heat generated in shearing contacts, minimizing friction, thereby minimizing the amount of heat the film must dissipate is also of importance. Highly ordered films demonstrate lower shear stresses than those with greater disorder,[193,194] due to the lower surface energy of a well-ordered, methyl terminated SAM, and the reduced potential for energy dissipation into a more rigid surface layer.[195,196] To better understand how these films behave, and fail, in MEMS, it is necessary to understand how formation on rough surfaces affects the defect densities within the films as well as their ability to passivate the underlying surface, as this speaks directly to their ability to dissipate contact loads and survive repeated impact and shear.

We have recently investigated the effects of nanoscopic curvature on the properties of alkylsilane SAMs by FTIR and AFM, where silica nanoparticles were employed to form surfaces with reproducible asperity structures. Here the radius of the particle could be tuned to alter the degree of surface curvature. When the nanoparticles are fused to a surface such as Si, they allow for the formation of surfaces with uniform asperity structure and afford a facile approach to generating films with controlled roughness, allowing for systematic study of asperity-asperity contacts. The assembly of organosilanes could be investigated on the isolated nanoparticles or on the rough surface formed from them. Using this approach, the effects of asperity curvature, chain length, and monolayer preparation

method were investigated to examine adhesion in asperity-asperity contacts[142] as well as the assembly and tribology of organosiloxane films on rough surfaces.[33] Order within the silane monolayers could be followed by the location of the methylene asymmetric stretch,[197] a peak that has historically been used to characterize alkane film ordering and which is sensitive to the presence of *gauche* defects in the film. Unfortunately, this approach is not entirely quantitative, nor is it sensitive to the substantial nanoscale heterogeneity that is typically present in alkylsilane monolayers owing to the inflexible and irreversible nature of silane bonding on silica surfaces.

To better understand these systems in molecular detail, here MD simulations have been employed to investigate the effects of surface curvature on the structure of alkylsilane films on surfaces with nanoscopic curvature. Following along our FTIR and AFM studies, simulated silica nanoparticles were used as model asperity surfaces, with surfaces modified with alkylsilanes based on our experimental determination of their packing density and assumptions regarding their distribution on the surface. While these simulations are intended for the purpose of understanding tribology of SAMs on nanoasperities, our conclusions may also be extended to similar nanoparticle systems capped with organic ligands. Organic capping layers are critical in nanoparticle and nanocrystal fabrication and isolation, they are necessary to prevent agglomeration but may also affect the properties of these nanoparticles. For example, the effects have organic capping layers have been investigated in thermodynamic models of fractionation processes.[198] Capping agents may also affect the electronic and catalytic properties of nanoparticle systems. The efficacy of nanoparticle and quantum dot catalysts, for

example, is extremely dependent on the morphology and passivating nature of the capping layers used in these systems. In metal nanoparticle catalysis, the capping layer restricts access to catalytic surfaces sites of the particles,[199,200] thereby moderating their catalytic activity. Likewise, in quantum dot catalysis, reactant proximity is more important and is directly affected by the morphology of the passivating layers on these particles.[201]

Molecular dynamics simulations have been used in the past to understand the structural and tribological properties of SAMs[100,106,202-206] and to study contact of interfaces at atomic and molecular length scales.[11,207,208] Studies include the determination of shear as a function of film packing density,[209] nanoparticle adhesion,[105,210] single-asperity friction,[163,187,206,208,211] as well as investigations of vapor phase lubrication systems[212] and the role of adsorbed moisture in nanoscale contacts.[101,107] In this work, MD simulations were employed to examine the conformational and morphological properties of alkylsilane SAMs on surfaces with nanoscopic curvature, in order to systematically study the effects of surface roughness on the molecular order and resulting structures of the passivating films, to aid as a guide to understanding their function in mitigating friction and wear at nanoscale asperity-asperity contacts

3.2 Methods

3.2.1 Computational Methods

All simulations were performed with the LAMMPS software package developed at Sandia National Laboratories[88] on Texas A&M University's IBM iDataPlex and p5-575 clusters. Simulated hydroxylation and functionalization of the substrates, as well as analyses of the simulation results, were performed by custom-made software and

MATLAB scripts, and visualization was achieved with the Visual Molecular Dynamics[213] software package.

3.2.2 Substrate Preparation

For the preparation of the silica substrates, a periodic bulk α-quartz structure was annealed and quenched as per a previously described method, using the free-body CHIK silica potential,[119] resulting in a 3-dimensional periodic amorphous silica structure. For the preparation of flat surfaces, periodicity in the z-direction was removed, so that the surface normal was parallel to the z-axis. For preparation of the nanoparticles, particle structures were cleaved from the bulk structure and periodicity in all directions was removed. To reduce computational burden and data storage, the core of the particles was removed leaving only a 15 Å thick shell to be simulated, with the innermost atoms held fixed to maintain the particle morphology. Hydroxylation of the silica surfaces was performed by a script which first hydroxylated all terminal, under-coordinated silicon atoms, followed by condensation of resulting hydroxyls by proximity, with parameters tuned such that the density of hydroxyl groups (reported in Table 1) and the ratio of Si-$(OH)_2$ to Si-OH groups were similar to previous results.[121] The surface was then relaxed by simulation for several picoseconds prior to functionalization. The final result is illustrated for a 7 nm diameter particle in Figure 1a. More details are provided in Appendix A.

3.2.3 Alkylsilane Functionalization

Initially all-*trans* configurations of octyl-, dodecyl-, and octadecylsilane were appended to the nanoparticle surface, as shown in Figure 3.1b. Molecules were placed

uniformly across the particle surfaces, maximizing intermolecular spacing. Adsorption was assumed to occur at surface silanol groups, such that every molecule is covalently bound to the surface via a siloxane bond. Proximity conditions similar to those used in preparing the hydroxylated silica surface were applied to condense remaining hydroxyl groups of the appended silanes, forming siloxane bonds between molecules or to hydroxyl groups on the particle surface. Molecules were appended at a coverage density of 1.5 molecules/nm^2 as indicated in Table 3.1, corresponding to results from TGA analysis from our earlier studies of silica nanoparticle functionalization,[33] unless otherwise stated. Images of the initial and final film structures are presented in in Appendix A.

Table 3.1. Molecular Coverage and Simulation Parameters.

Particle Diameter	Simulation Hydroxyl Density		Molecules to Achieve 1.5 Molecules/nm^2
	Count	OH/nm^2	
7 nm	678	4.4	231
12 nm	1951	4.3	679
40 nm	21289	4.2	7540
Flat Surface (A=341 nm^2)	1432	4.2	512

3.2.4 Simulations

All simulations were performed as fully atomistic simulations using the OPLS forcefield[118] with additional terms for the silica bulk.[214] Systems were initially relaxed using a soft cosine inter-atomic potential to eliminate destabilizing close-particle

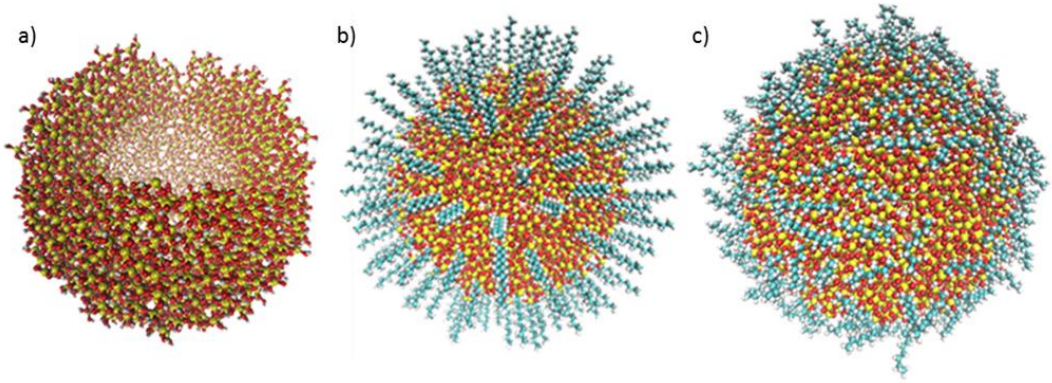


Figure 3.1. Demonstration of simulation preparation procedure for a 7 nm particle coated with octylsilane, beginning with (a), the hydroxylated particle surface showing the core removed and the 1.5 nm thick shell, followed by (b) functionalization with octylsilane in an all-trans configuration, and (c) after the simulation run time of 2.1 ns. More graphics of the simulated systems are provided in the supporting information.

interactions resulting from system construction.[214] Integration was performed using the RESPA technique,[110] wherein bonded interactions were calculated at 0.075 fs, angular and torsional interactions at 0.15 fs, and pairwise interactions at 0.3 fs. The system temperature was managed by a Langevin thermostat with a damping constant of 10 fs. The temperature of the systems was raised to 500 K for 90 ps, followed by cooling to 298 K at a rate of 6.73×10^{12} K/s. This was done to accelerate the equilibration of the system, and had been verified to show no different in the end result of the simulations. After cooling, the simulation was allowed to progress for a total of 6.4 million timesteps, for a simulation time of 2.1 ns, with an example endpoint shown in Figure 3.1c.

Snapshots of the simulations were taken every 9 ps, and evaluated for *gauche* defect density and distribution, percentage of exposed surface, and film thickness via homemade scripts, with simulation results reported as an average of the last 10 snapshots (90 ps).

Images of the structures at the beginning and end of the simulations are provided in the supporting information, as well as a video of one of the simulation trajectories.

3.3 Results and Discussion

3.3.1 Surface Coverage of Alkylsilane on Silica Nanoparticles

Prior studies using thermal analysis of the extent of surface functionalization by alkylsilanes on silica nanoparticles have shown that the packing density of silanes on the particles are relatively low (~1.5 molecules/nm^2).[33] Typical packing densities of alkylsilanes on flat silica surfaces have been reported to range from approximately 2-2.5 molecules/nm^2.[215] Optimal silane density has been observed to be as high as 4 molecules/nm^2 in other circumstances,[216,217] indicating that there already exists substantial free volume in these alkylsilane films. Such curvature dependent packing density of dodecyltrimethoxysilane on silica nanoparticles has also been observed by Feichtenschlager and co-workers on particles ranging from 10s of nanometers up to hundreds of nanometers, further demonstrating that there is a clear effect on film formation as a result of surface roughness.[32] The underlying silanol density typical of silica is 4-5 silanols/nm^2,[218] however due to the irreversible nature of silane bonding, reorganization of the film necessary to consume all surface binding sights doesn't occur as it would in more tightly packed SAMs like alkanethiols on gold.

The notable differences in silane coverage suggest that there are fundamental differences in the silanization chemistry on flat and nanoparticle surfaces. Extension of this low observed surface coverage from nanoparticles to the nanoscale asperities found on roughened surfaces is supported by excellent agreement of IR spectroscopic results

between both isolated functionalized nanoparticles (measured in KBr pellets) and surfaces with the same nanoscale roughness measured by IR transmission experiments.[33] As curvature itself naturally imparts greater free volume to the film, this low observed molecular coverage would seem to further exacerbate the impact of curvature on the integrity of the silane films, reducing the ability of alkylsilane SAMs to reduce interaction energies and dissipate contact forces. This decrease in surface coverage also paves the way for intercalation of other species, such as THF, hexane, water,[33] and 3P1P,[167] which have all been observed to intercalate into silane SAMs on rough surfaces while not doing so on similarly functionalized flat surfaces.

A root cause of the overall low molecular coverage of the alkylsilane films on the nanoparticle surface may be a result of an incompatibility between the natural film formation processes of alkylsilanes with curved surfaces. An island growth mechanism, attributed to preorganization of alkylsilanes[219] at or near the surface, has been observed on a variety of flat surfaces[89,220-222] due to the relatively high barrier to surface binding, which allows for surface diffusion and preassembly prior to anchoring of the molecules to the surface. Such preassembled aggregates will prefer planar geometries, and therefore be less likely to form on and bind to the curved surface of a nanoparticle in solution. Dynamic light scattering experiments of OTS micelles demonstrate typical radii of curvature of 200 nm, suggesting that achieving greater curvature is energetically unfavorable.[223] Additionally, large molecular aggregates that would form on flat surfaces will be less likely to form at the finite surface of the nanoparticles, where the available moisture to assist in surface binding is also more limited. Using geometric and structural

analysis, Stevens suggests that predominantly surface bound monolayers of alkylsilanes will saturate at approximately one third of maximum coverage,[224] similar to the coverages observed here. He likewise suggests the high coverages observed of trichlorosilanes is a result of minimal surface binding, as suggested by FTIR spectroscopy of these OTS SAMs,[225] and rather interactions with interfacial moisture layers allow for more favorable tight packing configurations, as observed in high resolution AFM of SAMs prepared by the Langmuir technique. Such moisture layers may not be stable on the curved nanoparticle surfaces, rendering direct surface binding as the primary functionalization route. As there is sufficient evidence that the island formation mechanism observed on flat surfaces may not be transferable to nanoparticle surfaces, and owing to the difficulty in accurately modeling such a mechanism, the functionalization scheme used herein assumes no preorganization of the film molecules, which are applied uniformly to the silica surfaces so as to maximize intermolecular spacing.

3.3.2 Chain Conformations and Heat Dissipation Potential

Self-assembled monolayers with tight packing reinforce conformational modes into all-*trans* configurations of the molecules, directed relatively normal to the functionalized surface. As we consider the tribological effectiveness of these films on surfaces with curvature, it is natural to consider how roughness effects both the conformational reinforcement via tight packing and the general orientation of the molecules on the surface, as these features speak directly to the film's ability to dissipate contact load and heat generated from shear. The role of *gauche* defects on the contact mechanics and wear reducing properties has been extensively considered. Harrison and co-workers have

demonstrated, by simulation of alkane films on diamond surfaces,[172] that formation of terminal *gauche* defects in the film as a means by which films can accommodate pressure from a flat counterface. Salmeron[171,226] has likewise considered extensively the pathways of load dissipation in alkane films, considering how chain tilt and the formation of both terminal and internal *gauche* defects play a role in this process. Finally, Soza *et al.* demonstrates the formation of defects in tetracosane mono- and bilayers on a graphitic substrate after simulated scanning of an asperity across their surfaces, and observed that friction is reduced upon artificial stiffening of the molecular torsions.[195] This latter point implies that a more rigid film will produce lower friction by reducing energy dissipation into the surface, which is observed for more well-ordered, rigid SAMs.[196,204,227,228] Defect formation, the conformational change from a *trans* methylene unit to a *gauche* methylene unit, is expected to increase on surfaces with curvature owing to the fact that, as the chains extend away from the surface, they must fill a greater volume. *Gauche* defects fill more volume than *trans* configurations, and will naturally form so as to maximize intermolecular interactions. Their formation is also a mechanism by which contact loads and heat generated by shear may be reversibly dissipated, therefore their population in the freestanding film is of considerable importance, as it speaks to the availability of energy dissipation pathways, the effectiveness of these pathways, and the overall molecular rigidity of the film.

With these facts in mind, *gauche* defect densities were examined as a function of surface curvature and chain length, as well as distance from the surface, and in a later discussion, as a function of packing density. Figure 3.2 depicts the progression of defects

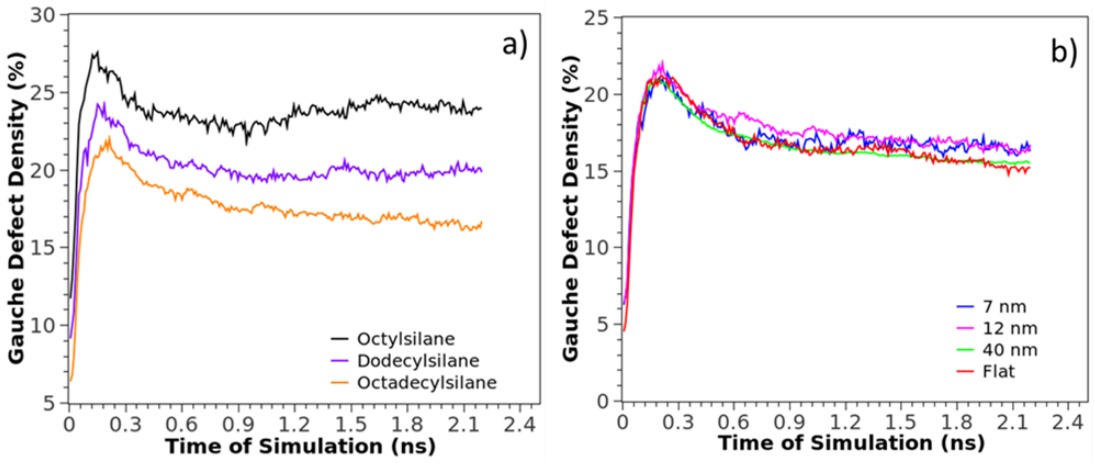


Figure 3.2. Simulation trajectories showing the progression of *gauche* defect densities for (a) OTS coated particles of all curvatures studied and (b) 12 nm particles coated with all alkylsilanes studied.

in the films over the course of the simulation for surfaces with varying radii of curvature. The initial spike in defect density was a result of the initial annealing step, and was followed by equilibration of the films. Defect densities as both a function of particle curvature and chain length, measured at the end of the simulations, are shown in Figure 3.3a. Figure 3.3b demonstrates the corresponding methylene asymmetric stretch peak location observed experimentally by FTIR spectroscopy, which has long been used as an indicator of defect density in SAMs.[229] These figures are scaled so that their maxima and minima approximately correspond to fully disordered liquid polyethylene[230] and fully ordered n-alkane systems respectively.[231] Data points generated from higher coverage flat surface simulations are shown as well, as higher packing densities are consistent with films on flat surfaces (*ca*. 2-2.5 molecules/nm^2, compared to 1.5 molecules/nm^2 observed on nanoparticle surfaces). As expected, the chain length dependence was considerable in

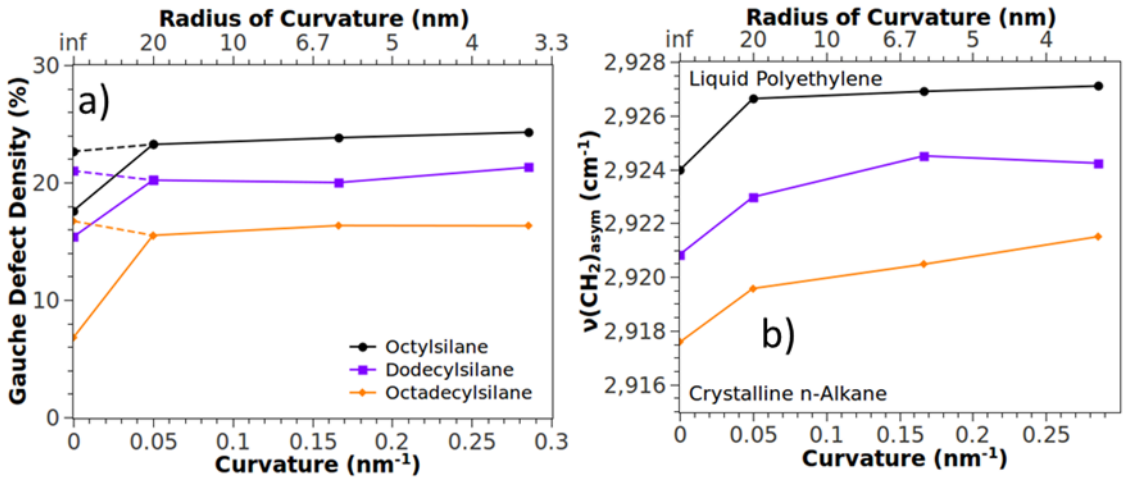


Figure 3.3. Percentage of *gauche* defects from the simulations as a function of curvature (a). For reference, a corresponding plot of the frequency of the asymmetric methylene stretch is shown, scaled to similar extremes corresponding to a crystalline n-alkane monolayer (ν=2915 cm^{-1}) and liquid polyethylene (ν=2928 cm^{-1}). The points at zero curvature connected by dotted lines in (a) correspond to flat surface simulations at film packing densities of 1.5 molecules/nm^2, while the solid lines correspond to packing density of 3.0 molecules/nm^2, as this is in the range of typical packing densities of alkylsilanes on flat surfaces. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

both the experimental and simulated systems. This agrees with observations on both flat[232,233] and curved[234] surfaces, and a corresponding reduction in friction[38,197,235] for a variety of alkane based SAMs. Curvature dependence, on the other hand, is much less dramatic. By comparison, SFG studies on alkanethiol capped gold nanoparticles demonstrate relatively little change in film disorder between flat and curved surfaces until the radius of curvature approaches 2 nm.[236] It is important to note the differences between these two systems. Contrary to the observed decrease in packing density of alkyltrichlorosilanes observed here, alkanethiols are observed to pack more tightly on gold nanoparticle surfaces,[237,238] suggesting that curvature effects compete with increasing

molecular forces to affect no net change in defect density. It has also been observed that dodecyltrimethoxysilane packing density continues to vary with curvature for larger radii particles,[32] which suggests that packing density may continue to decline as curvature increases, such that the spectroscopically observed increase in disorder with curvature may in fact be coverage driven.

The spectroscopic data also indicates that *curvature dependence* increases with chain length in the spectroscopic data, while the simulations demonstrate consistent, minimal curvature dependence across chain lengths. This may be a result of the uniform coverage assumption applied in the construction of the simulated systems. The uniform coverage assumption becomes less valid as the island growth mechanism becomes more relevant, and the preorganization that facilitates island growth is likely a chain length dependent phenomenon. Longer chain films will also be more capable of accommodating intercalated solvent molecules that can stabilize the film, increasing the observed order. Hexane, for example, is a commonly employed solvent for alkylsilanization, and was used in the functionalizations of particles examined spectroscopically. Intercalation of hexane in particular can both affect the disorder within the film, and experimentally it may interfere with the indicative methylene asymmetric stretch peak owing to its nearly identical chemical structure to the film, these effects are currently under investigation. Additionally, as discussed earlier, if packing density is in fact curvature dependent,[32] the effect may vary with chain length, resulting in greater curvature dependence for the longer chain length films.

As a function of distance from the surface, the observed distribution of defects did not

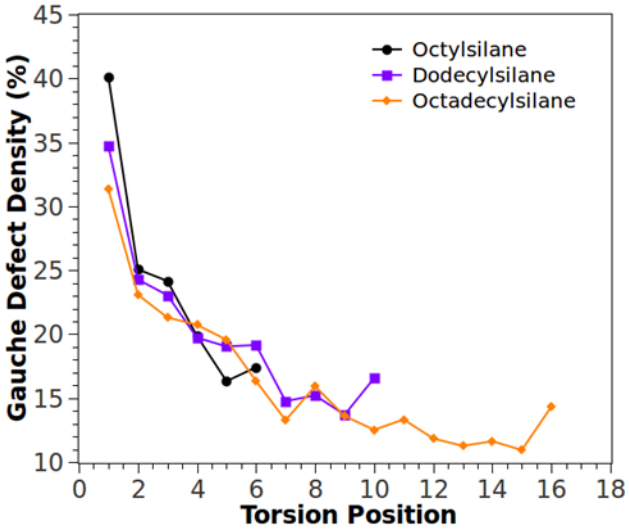


Figure 3.4. Percentage of *gauche* defects as a function of location away from the surface. Position 1 indicates the Si-C-C-C torsion closest to the surface, with increasing position corresponding to torsions located further up the chain. The majority of defects appear close to the particle surface indicating the molecules are driven to pack at their extremities with their nearest neighbors. The data shown is for simulations on the 12 nm particle for all chain lengths. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

coincide with the expectation that defects would increase as the chains project away from the surface, where free volume would typically be greater,[239] rather, the greatest defect densities occurred close to the particle surface, this is clearly demonstrated in Figure 3.4, and was observed for all chain lengths and surface curvatures studied (see supporting information). Interestingly, the molecules appeared to distort and bend over to maximize intermolecular interactions with their nearest neighbors and the particle surface. Their choice of interaction, with each other or the surface, is dependent on chain length and surface coverage, as well as the way the molecules are distributed across the surface, shorter chain molecules tend to interact with the surface, while longer chain molecules

interact with each other. This is another situation where the uniform coverage assumption will ultimately impact the final result, as preorganized film molecules are more likely to interact with each other than the particle surface, and sterics would force the majority of defects to the termini of the film molecules.[171] This would result in greater ordering within the films, but at the cost of a substantial increase in the apparent surface energy of the interfaces due to the increase in exposed silica surface, likewise resulting in impaired wear resistance and film stability.

While the direct measurement of *gauche* defects is indicative of the availability and energetics of dissipation pathways, it doesn't necessarily imply that defect formation is an accessible pathway. Film tilt has been observed to play a considerable role in load dissipation,[191] and it is therefore reasonable to assume that a more upright film will be better able to dissipate contact pressures. Molecular tilt angle, typically measured directly from the anchor point of the molecule to its termini,[105,240] is often use to gauge this, however due to the highly irregular structure of the molecules, orientation was examined by considering the average orientation of the methylene units as a function of position in the chain, depicted in Figure 3.5. Correlation to chain length was strong, while curvature effects are not discernible. Additionally, an important chain length dependent transition was observed. For OTS, the orientation away from the surface normal did not appear to monotonically increase as the chains extend away from the surface, as it did for the shorter chain lengths. This effect corresponds to the molecules lying down on the surface, and therefore indicates that the OTS molecules were effectively able to "find" each other, allowing intermolecular interactions to play a greater role in stabilizing the molecules in

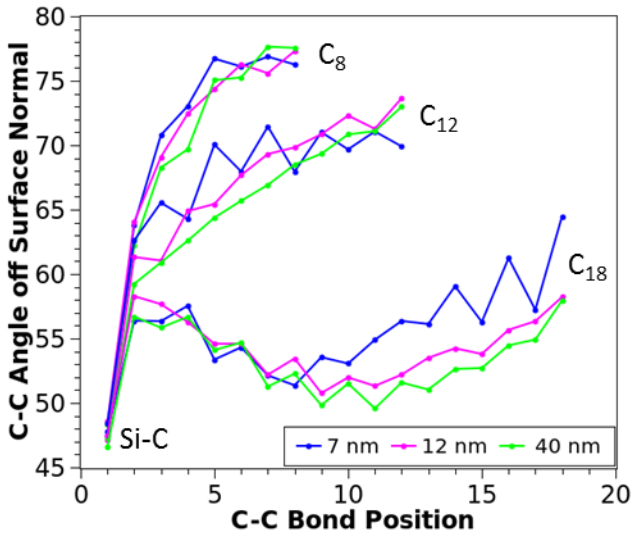


Figure 3.5. As a function of distance up the length of the chain, the mean angle of the C-C bonds with respect to the surface normal of the particle is shown, with the initial Si-C bond indicated at position 1. While octyl- and dodecylsilane appear to monotonically increase, there is a clear decrease in methylene angle in the case of OTS within just a few methylene units from the surface, indicative of the islanding effect that leads to the visual appearance of aggregation of the molecules on the surface. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

trans configurations directed away from the particle surface, and is visible in the equilibrated film structures (see supporting information). This feature would be expected to be a product of mean molecular spacing, related to the packing density, and will be discussed later.

From these results, it is clear that the films demonstrated relatively high defect densities and a tendency for the molecules to lie down, suggesting little to no reinforcement or accessibility of the conformational modes that would assist in the dissipation of contact forces. This is clearly demonstrated by AFM adhesion measurements,[176] where adhesion was observed to be consistently higher for a

functionalized AFM tip in contact with an Si(100) surface, compared to an unfunctionalized tip against a functionalized Si(100) surface, with greater adhesion an indicator of greater dissipation in the contact when the SAM is on the asperity surface compared to the flat surface. It does not appear, however, that curvature is substantially responsible for this effect, as the observed curvature dependence in the simulations was relatively weak. The spectroscopic results indicate substantially greater curvature dependence on the molecular ordering of the films and their tribological performance, and to better understand why this might be, one can consider the known deviations between experiment and the simulated systems. The most obvious deviation is the environment, these simulations were performed in a vacuum on pristine, functionalized particles, whereas the real systems were studied under ambient conditions, and intercalates are known to be present[33] in the films or may be introduced.[167] Intercalates can have the effect of affording greater effective packing densities within the freestanding films, improving the projection of the films away from the surface and reinforcing energy dissipation pathways. The effect of intercalates on the tribology of these films, however, is not uniform, suggesting that other factors are important. For example, hexane is known to intercalate into films when used as a functionalization solvent.[33] 3P1P may likewise be intentionally intercalated into films.[167] Films intercalated with the latter, however, demonstrate substantially greater tribological performance, likely owing to a greater affinity of the 3P1P molecules to the surface via hydrogen bonding and the film via improved van der Waals interactions. Film packing appears to be a critically important aspect, and a discussion of the effects of packing will follow.

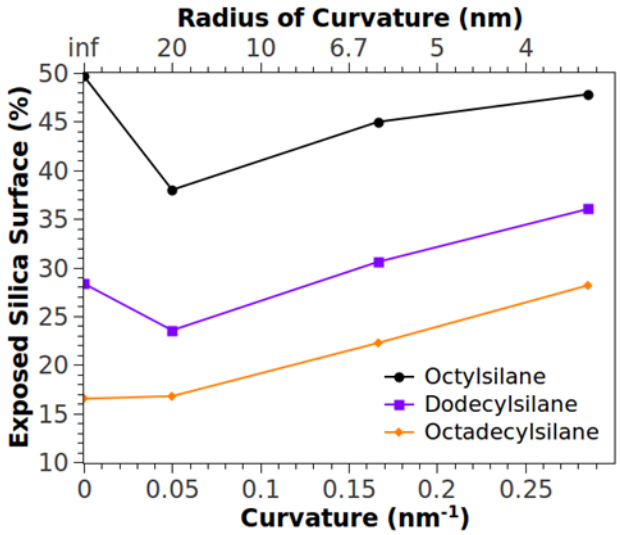


Figure 3.6. Percentage of exposed surface on the particle surfaces for all curvatures and chain lengths studied, demonstrating a strong relationship between coverage and curvature. The flat surfaces appear as a deviation from the trend due to differences in the way the surface mesh is defined for flat and nanoparticle surfaces. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

3.3.3 Surface Protection and Passivation

The ability of the film to passivate the underlying silica surface is critical, as exposed silica surface presents high energy surface sites that facilitate rapid degradation of the surfaces in contact and provide active sites for water binding, which may catalyze the stress-induced scission of siloxane bonds on the surface.[241] Notably, the low molecular coverage led to a clear aggregation of the molecules, particularly for longer chain lengths, resulting in a large fraction of exposed silica surface accessible to the environment and interfacial contact. Surface coverage was quantified by forming a spherical mesh about the particle and identifying mesh points as exposed if there were no hydrocarbon atoms within a 20˚ cone directed normal to the surface. This is summarized in Figure 3.6,

wherein a clear curvature and chain length dependence was observed. Chain length dependence was expected, owing to the greater amount of material on the surface for greater chain lengths, resulting in greater surface coverage. The percentage of exposed surface also demonstrated distinct curvature dependence, indicating that on nanoasperities, the surface is more accessible to interactions with the environment. This is perhaps most clearly evident in time-dependent AFM adhesion studies on both flat and asperity surfaces functionalized with octadecyltriethoxysilane.[176] In these studies, the approach/retract rates were varied, and the resulting adhesive force measured. No significant change in adhesive force occurs for the functionalized flat surfaces as a function of rate, while there appears to be a slow timescale change in adhesive force for both a tip approaching a functionalized asperity surface, and a functionalized tip approaching an unfunctionalized surface. This slow timescale process is likely a result of bridging siloxane bond formation between the tip and the substrate, owing to the extent of exposed surface present. These interactions could include interfacial interactions in asperity contacts, increasing adhesive and shear forces within MEMS devices, as well as interactions with molecules at the particle surface. Alkylsilane functionalized nanoparticles have in fact been observed to uptake solvent molecules during functionalization,[33] and can be induced to uptake other molecules,[167] a feature that is not observed of flat surfaces. This is a critical finding and speaks directly to the failure of these films. Not only are the contact pressures achieved at asperity contacts the most threatening to surface integrity, but the ability of a SAM to passivate these asperities is extremely compromised as indicated by the extent of unprotected surface. This suggests

that, to achieve any meaningful surface protection at surface asperities in MEMS, efforts would need to be made to both reduce surface roughness and increase the passivating capability of applied SAMs, as it is unlikely that sufficient surface passivation could be achieved at the sharpest asperity contacts given the greater challenge of protecting sharply curved surfaces. Reducing surface roughness must be done carefully however, as substantial increases in real contact area will also negatively impact the frictional and adhesive characteristics of interfacial contacts within devices.

Passivation of surfaces with nanoscale curvature is not only desirable for tribological applications, but has critical implications in the general field of nanoparticle chemistry, including synthesis and nanoparticle catalysis. Sparse surfactant density is actually beneficial in metal nanoparticle catalysis, where the surfactants are needed to prevent particle agglomeration, but which restrict access to catalytic surface sites.[199] Islanding and aggregation of such films would ultimately improve catalytic activity, as film molecules interacting with each other will not interact with, or block, these catalytic surface sites. In quantum dot catalysis as well, the particle capping layer restricts access to the particle surface, controlling the mean proximity of substrate molecules and therefore moderating the catalytic properties of the quantum dot. Weiss *et al.* have demonstrated that the capping layer in a quantum dot/polymer matrix can be used to control the rate of photoinduced electron transfer.[201] In particular, they observed that the catalytic time constants depended on not only the length of the capping agent, but also on the morphology of the capping layer. They observed that corrections needed to be made for the collapse of the ligand shell, a property which is directly related to the amount of open

volume on the particle surface. They further surmise that islanding and aggregation of molecules in the passivating films renders the particles inaccessible to the polymer matrix in which they are embedded, results that are not unlike what is observed here, where aggregation of the surface bound OTS chains is qualitatively observed, distinct from the shorter chain films.

3.3.4 Surface Coverage Effects

Due to the apparent impact of nanoscale roughness on the coverage of molecules on the surface, and to provide perspective on the relevance and impact of roughness in applications of these films in devices, the effects of coverage were examined systematically in the absence of surface curvature. This has been considered before on flat crystalline surfaces regarding the shear stresses applied to a film in single asperity contact[227] and for different packing configurations of alkanethiols on the Au(111) surface.[194] The effects of packing density have been extensively studied on a variety of systems, including alkanethiols on the Au(111) surface,[193] alkane molecules tethered to a diamond surface,[204] and alkysilanes on crystallane silica.[209] It is commonly observed that disorder within the films increases as packing density decreases, and as a result, friction increases. There are two means by which increased disorder may increase friction: chemical effects, whereby the more disordered film has higher surface energy; and what may be described as kinetic effects, whereby dissipation is more favored in the less rigid, more disordered film.[195,196] Here packing is considered on flat, amorphous silica, where no underlying crystalline structure guides the packing of the film, using the same, uniform coverage assumption that was used in functionalization of the particle surfaces. It is

important to note that OTS films on flat surfaces display considerable heterogeneity, with high and low density regions readily apparent by AFM imaging.[220] It is therefore difficult to describe an OTS film in its entirety by MD simulation, and such simulations only provide insight into local properties of the films, not their bulk characteristics as would be measured in spectroscopic or large contact area tribological experiments. For well-ordered, crystalline regions of the film that form early in the film formation process, aided by surface moisture, high resolution AFM measurements have determined that the packing density can reach 4 molecules/nm^2,[216,217] while in lower density regions the coverage is

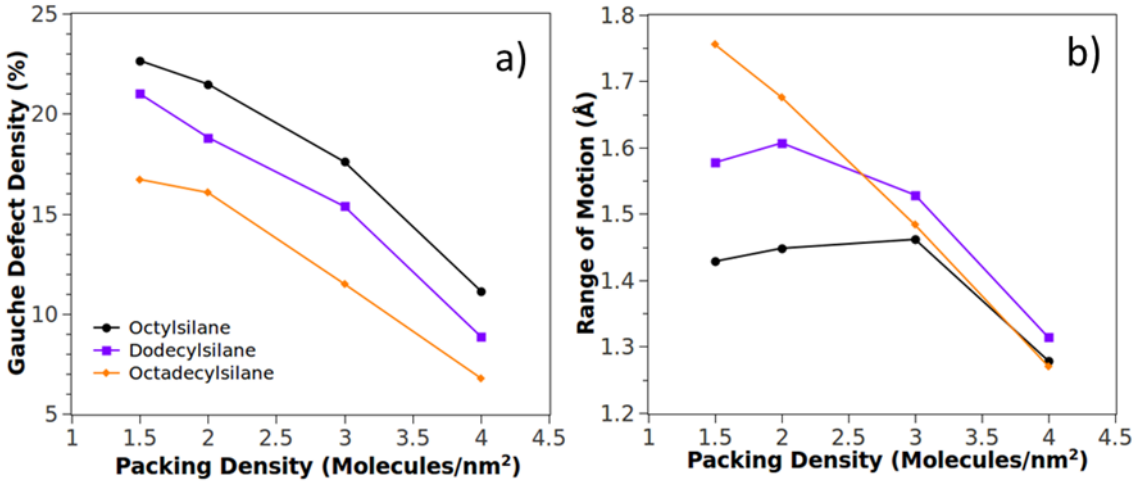


Figure 3.7. Percentage of *gauche* defects (a) and range-of-motion (b) for flat simulations as a function of molecular packing density. Film quality very clearly improved by increasing the molecular packing, owing to the corresponding decrease in free volume within the films. Range-of-motion, an indicator of molecular rigidity. At high packing density, an inversion as a function of chain length is also apparent, indicating a point at which sterics dictate molecular motion more than molecular degrees of freedom. Note: The typical packing density of trichloroalkylsilanes is 2-2.5 molecules/nm^2 on flat silica surfaces. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

not well-known but is likely considerably less. Here, the coverage is varied from the 1.5 molecules/nm^2 as observed for the functionalized nanoparticles, up to 4 molecules/nm^2, on simulated flat amorphous silica.

Figure 3.7a demonstrates the defect density dependence with coverage in the films. OTS films trend towards much higher degrees of order, with as low as 7% *gauche* defects at maximal coverage, less than half the observed disorder in the lower density films. Because the silane binding locations were artificially dictated by the location of hydroxyl groups on the surface, and not by their optimal packing, this number represents an upper estimate of the extent of disorder in an optimally packed OTS film. The reduction of defects is expected to play a role in reducing the friction of these films, and molecular rigidity has also been observed to play a role, with less rigid surfaces demonstrating increased friction.[195,242] To characterize the rigidity of the monolayers, ROM calculations analogous to that used by Harrison and coworkers[243] were employed to study the degree of molecular rigidity. In this analysis, the deviation of the vector connecting the anchor and endpoint of the molecules is averaged over all molecules and over the final 10 snapshots of the simulations (See supporting information), corresponding to the final 90 ps as with other the other measurements presented. Figure 3.7b shows the results of these calculations as a function of coverage. At low coverage, the ROM is greatest for the longest chain molecules, even though defect calculations would suggest they are more ordered. This is likely due to the greater molecular degrees of freedom in the longer chain molecules. These two effects therefore compete in the overall frictional response. As the packing density increases, the ROM of all three chain lengths converges to a minimal

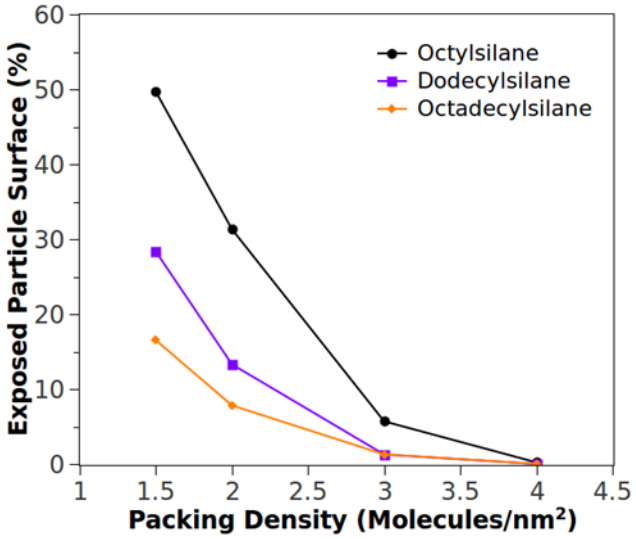


Figure 3.8. Surface exposure as a function of packing density. It is clear and expected that, with more molecules on the surface, the surface is better protected from interfacial silica contacts and the adsorption of moisture. Lines are included to guide the eye and do not indicate a functional relationship.

value, and is lowest for the longer chain systems. In this regard, both decreasing defect densities and increased film rigidity will conspire to reduce the friction of these interfaces. Curvature effects on ROM are presented in the supporting information, and are primarily dominated by the packing density effects observed here, with behavior similar to that demonstrated here at low coverage.

The extent of exposed surface, indicated in Figure 3.8, declines very clearly with packing density. This is not surprising as with increased packing density comes more material and saturation of surface sites. This is also in good agreement with the fact that OTS films on flat surfaces are relatively impervious to intercalates. Additionally, Figure 3.9 depicts the methylene group orientations as a function of position and orientation. As

packing density increases, uniformity in the chain tilt and clearly defined odd-even effects are observed, with all chain lengths converging to a mean tilt angle of ~37˚ at 4.0 molecules/nm^2. The tilt angle of alkane SAMs is a distinctly packing density dependent property, as the tilt is a result of the molecules optimizing van der Waals contact between the molecules.[244] OTS films on formed on flat surfaces have been observed to have a tilt angle of ~20˚,[245] suggesting very high packing densities, in agreement with the observed 4.4 molecules/nm^2 packing density observed by high resolution AFM of OTS on silicon oxide.[217]

From this, it is easy to conclude that a substantial challenge that must be addressed in the application of SAMs as wear reducing layers on surfaces with nanoscopic curvature

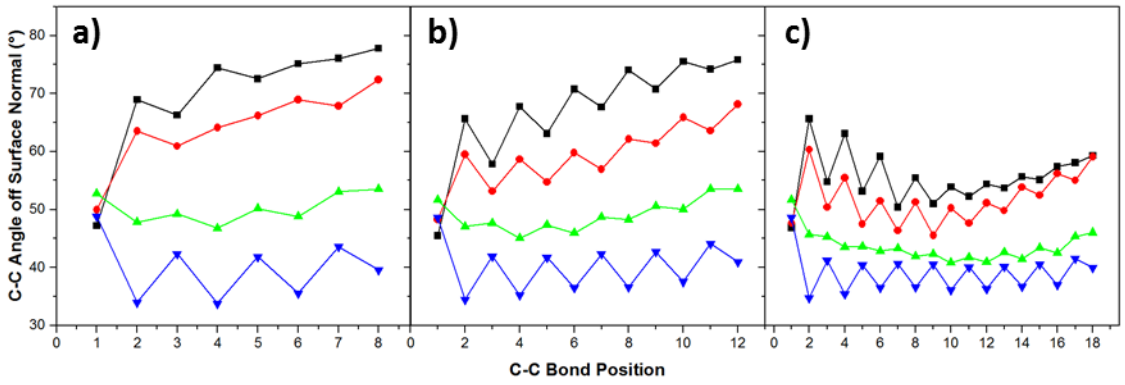


Figure 3.9. Mean C-C bond angles relative to the surface normal of a flat, periodic silica surface for octylsilane (a), dodecylsilane (b), and octadecylsilane (c) at 1.5 (black), 2.0 (red), 3.0 (green), and 4.0 (blue) molecules/nm^2 packing density. At low coverages, the shorter molecules appear to lay down on the surface, indicated by the roughly monotonic increase in the mean angle. As packing density increases, however, these short molecules behave very similarly to octadecylsilane. Lines are included to guide the eye and are not meant to illustrate a functional relationship.

is that of packing density, though this challenge is not easily overcome. This problem is naturally connected to surface curvature, as curvature has the effect of decreasing packing density away from the surface, but it would appear that the natural packing density observed on asperity surfaces is too low for curvature effects to be substantial. It has long been understood that moisture is a critical element of alkyltrichlorosilane film formation,[246-248] and adsorbed surface moisture specifically is needed for hydroxylation and assembly of highly ordered alkylsilane domains.[249] On a flat surface, multilayers of water can exist to facilitate binding and polymerization of the silanes at the surface. This is not the case on nanoparticle surfaces, where a finite surface area is present, and therefore a finite amount of moisture is available for film formation. On rough surfaces, as would be found in MEMS, formation of large, contiguous multilayers of water may likewise be inhibited by the irregularity of the substrate, with potentially greater availability in cavities rather than at the apex of asperities where robust film formation is most needed. Additionally, the lower availability of surface moisture would lead to a preference for the silanes to bind directly to the surface, which has been shown to lead to poor packing density owing to steric effects, with expected coverage of about one third of a complete monolayer expected, in agreement the coverage observed on the particle surfaces.[224] On rough polysilicon surfaces, it has been proposed that crystalline, tightly packed film formation primarily occurs at grain boundaries, presumably where water adsorption is favored.[250] This general lack of control over the film morphology likewise makes reliable protection of nanoasperities a difficult task that must not be neglected.

3.4 Conclusions and Outlook

Several key conclusions can be drawn from these studies. First, the low surface coverages observed for silanes on nanoasperities suggests a key route to the loss of the films typically observed on these asperity-asperity contacts. The low packing densities leave exposed surface that provides easy access pathways for water which can catalytically degrade the surface films. Additionally, the reduced surface coverage leads to a reduction in film integrity due to the lack of nearest neighbor stabilizing interactions that would otherwise reinforce conformation modes of energy dissipation and increase the rigidity of the film, reducing the overall dissipation, and friction, in the system. Our work demonstrates that special care must be taken in the application of SAMs on rough surfaces, particularly those found in MEMS devices. It is clear that the silanization chemistry, which is already extremely sensitive to environmental conditions, is also sensitive to the morphology of the surface. This is likely a surface moisture effect, and attempts to develop less moisture dependent functionalization approaches[169,251,252] will be crucial in developing complete, well-ordered monolayers on MEMS device surfaces.

If the challenges of sufficient passivation are solved, however, questions still remain. It is not clear if complete passivation of MEMS device surfaces would still lead to effective surface passivation. Due to their static nature, if wear of the interfaces is not eliminated, SAMs will still be incapable of providing lasting lubrication in sliding contacts of MEMS devices, and indeed any tightly bound molecular film would be predicted to fail. This has motivated two primary tracks of research in our group. First, using the model systems developed in this work, we will study how the contact mechanics and friction of asperity

contacts change as a function of molecular packing density. The simple question we seek to answer is, if the coverage challenges demonstrated in this work are indeed solved, will effective lubrication and wear resistance be within reach. Approaching this problem with simulation is ideal due to the substantial investment that would likely be required to develop both develop a solution to the coverage challenges presented, and to apply these solution to MEMS devices.

The high wear resistance and low friction of vapor phase lubrication schemes has motivated our second line of inquiry. It has been found that MEMS tribometers demonstrate low friction and stable operation when immersed in a pentanol vapor,[25] and wear track formation in AFM experiments is likewise inhibited.[160] We have found that similar results may be achieved by infusing OTS SAMs with a low vapor pressure alcohol, 3P1P.[167] Both low friction and stable surface protection have been demonstrated. The benefit of this approach is that no modification to the device is required to support the mobile lubricant. Due to the complexity of the system, however, the mechanisms of lubrication and wear resistance are unclear. It has been observed that 3P1P binds to exposed surface sites in the low coverage film, suggesting improved passivation of the silica surface, but during sliding contact under high loads, where OTS would fail on its own, it is not clear if the OTS remains or is sheared away, with lubrication achieved by 3P1P alone. The wear resistance of the SAM may be enhanced by the infusion of 3P1P, which will result in an increase in the effective packing density of the SAM. These questions will be addressed by both MD and local spectroscopic approaches like Tip-Enhanced Raman Spectroscopy, and with mechanistic insight further improvements to the

combination of mobile and static lubricants may be proposed.

Understanding the means by which molecular lubrication is achieved at the nanoscale for both static and mobile molecular films is key to developing and improving these lubrication approaches for dynamic MEMS. Much success has been made, and full realization of the capabilities of these devices may soon be a reality. Our work provides a better understanding of the deficiencies of SAM-based lubrication strategies, suggesting both improvements and acting as a step towards understanding more complicated, mixed component systems that will likely be more successful. Future works will focus on understanding the mechanics of these systems, and how improved mixed component systems act to achieve more effective lubrication.

CHAPTER IV

UTILIZING ATOMISTIC SIMULATIONS TO MAP PRESSURE DISTRIBUTIONS AND CONTACT AREAS IN MOLECULAR ADLAYERS WITHIN NANOSCALE SURFACE-ASPERITY JUNCTIONS: A DEMONSTRATION WITH OCTADECYLSILANE FUNCTIONALIZED SILICA INTERFACES[*]

4.1 Introduction

The forces of friction and adhesion are incredibly important in the operation of machines and devices, affecting device longevity and energy efficiency with enormous aggregate impact on the economies of the world.[2] Because of this, substantial research and development has focused on the minimization of friction and control of these forces. More recently, to achieve greater performance in the most demanding applications, research has focused on the underlying mechanisms of friction.[253,254] The laws governing the observable friction response are well known. For macroscale contacts Amonton's law applies, for which friction is proportional to normal load regardless of contact area.[3] For dry, single-asperity contacts, friction is proportional to contact area,[255] and explanations have been offered which merge these two perspectives.[23,256] From a mechanistic standpoint, various pathways of dissipation have been identified, including strain relaxation,[257] phonon[258] and electron-phonon interactions,[259] and the expression of

[*] Reproduced with permission from Ewers, B. W.; Batteas, J. D. "Utilizing Atomistic Simulations to Map Pressure Distributions and Contact Areas in Molecular Adlayers within Nanoscale Surface-Asperity Junctions: A Demonstration with Octadecylsilane Functionalized Silica Interfaces". *Langmuir*. **2014**, DOI: 10.1021/la500032f. Copyright 2014 American Chemical Society.

tribochemical reaction pathways at surfaces. A significant challenge that remains is to apply this knowledge to relevant lubrication applications, to develop coatings designed to interfere with these dissipation pathways and to optimize their effect for applications under the most extreme conditions.

The factor with the greatest impact on the development of a lubrication scheme is the sliding regime in which the device can and will operate. Dry sliding is generally the least optimal circumstance. Here, all energy dissipation occurs at the interface of the contacting solids. Strains can be quite large and, depending on the surface energies involved, considerable tribochemistry and material transfer can occur, giving rise to higher friction and more substantial wear. At the other extreme, hydrodynamic sliding is a much more ideal situation, wherein dissipation occurs only at the boundary between the surface and the lubricant film, and within the lubricant fluid itself. Though hydrodynamic sliding is preferable, it is not always achievable nor is contact between the surfaces unavoidable.[260] Boundary lubricant additives are often employed in motor oils,[261] for example, as a last line of defense when parts do come into contact. Furthermore, some devices are simply not amenable to hydrodynamic lubrication; for example viscous drag prevents the operation of MEMS in lubricant fluids[262] and space based applications are simply not suited to liquid lubrication. In these cases, surface coatings and boundary lubricants are the only means of lubrication, and a variety of options have been investigated including SAMs,[141] layered or nanostructured materials like graphene[263] and molybdenum disulfide,[264] polymers,[265,266] and ionic liquids.[267] In all cases, surface roughness is an important factor that must be considered. Surface roughness is inevitable, but it is also

necessary to minimize the aggregate surface forces at contacts. Unfortunately, roughness magnifies the pressures where contact does occur, demanding exceptional performance of boundary lubricants and coatings. To achieve these goals, it is necessary to understand fundamentally the mechanics of lubricant coatings at the atomic scale and in relation to all the relevant chemical and mechanical factors that may be present in the desired applications.

There are two key factors that dictate the performance of a boundary lubricant, chemical passivation of the interface[39] and mechanical decoupling of the bulk material.[208] Chemical passivation is necessary because technologically relevant surfaces, mostly metals and metal oxides, have high surface energies and offer a vast array of tribochemical pathways for energy dissipation. Even relatively inert materials can become chemically active due to the exposure of dangling chemical bonds during wear. These chemical pathways naturally lead to the degradation of surfaces and wear of the devices. In most MEMS devices for example, the silicon from which they are fabricated is naturally terminated by a silicon oxide film.[141] When these surfaces are pressed into contact, the formation of siloxane bonds under applied pressure can lead to greater energy dissipation during sliding. Boundary films ideally terminate these chemical pathways by reducing the surface energy of the interface. Self-assembled monolayers have garnered interest as protective coatings in part because many of their dissipative pathways are reversible,[171,172] via conformational changes in the ordering and packing of the film at the surface. Alternatively, self-healing films from vapor phase lubricants[25] and boundary lubricant additives[268] can handle the dissipation of energy in a sacrificial manner, as they are

constantly replenished at the interface. In order to prevent surface degrading tribochemistry, the boundary film must effectively shield the two contacting surfaces from direct interaction, and the ability to characterize this passivation effect is essential to the design of such protective films.

Mechanical decoupling of the surfaces results from decreased real contact area, which nominally reduces the strains experienced during sliding. This reduction of contact area must be defined carefully, though. Soft coatings on hard surfaces, for example, increase the overall area of interaction, but decrease the contact area between the underlying surfaces. It is the contact of the underlying surfaces that is most critical, as these surfaces are most directly coupled to bulk dissipation pathways and for which tribochemistry is most undesirable. A boundary layer that effectively broadens the distribution of pressure and blocks direct contact pressure between the contacting interfaces will therefore be most effective.

Understanding the impact of boundary lubricant molecules on the distribution of contact and pressure at an interface is not only essential for the development of effective lubricants, it provides insight into macroscopic friction in all cases. Technologically relevant surfaces are almost always coated with something, be it an oxide layer, moisture, adventitious hydrocarbons, or tribofilms[269] formed during sliding. Rarely is there a situation in which the chemistry and mechanics of the interface closely resembles that of the bulk. To understand friction between real surfaces it is necessary to understand how pressure and strain is transmitted through these coatings. Furthermore, it provides insight into the electrical and thermal conductivity of nanoscale contacts relevant to the

development of microdevices, where environmental conditions can lead to similar surface contamination and alteration.[270,271] Continuum and finite element modeling can be of some assistance, but the mechanical properties of boundary layers are often not well known, either because they are formed during sliding contact, are the result of surface contamination, or are simply too thin to maintain their bulk characteristics. The Thin Coating Contact Model,[272] for example, is effective for coatings that are much thicker than the penetration depth of contact, but it is ineffective for coatings such as molecular monolayer or films that are only a few atomic layers in thickness.

Because of these limitations, MD Simulation is a much more effective tool to explore the properties of boundary films and molecular surface coatings, and it has been used extensively to examine the friction response of coatings like self-assembled monolayers,[172,240,273-275] as well as graphene,[276] amorphous carbon coatings,[277] and vapor phase lubricants.[212] Atomistic simulations also provide tremendous insight into the dissipative mechanisms at the atomic and molecular level.[172] Unfortunately, a parameter as simple as contact area is difficult to define at the atomistic limit and on the timescales of atomic motion.[181] Something as seemingly straightforward as what constitutes two atoms in contact is non-trivial. While one might wish to define atomic contact as repulsive interaction, this ignores the weak van der Waals bond formation and scission processes that are certainly common and relevant dissipative processes. As a result, various methods for measuring contact from atomistic simulations have been employed. Integrating the atomic forces through space for example, can produce pressure distributions that characterize the area of contact at the interface.[182,278,279] This approach is useful for

relatively large simulated contacts, where the large number density of atoms minimizes the effects of thermal fluctuations. An alternative and relatively straightforward approach is to define the two-atom contact either by force or proximity between the atoms. Contact area can then be defined by associating area with each atomic contact, an approach that is particularly feasible on crystalline surfaces, where each surface atom occupies a well-defined area.[177]

A relatively new approach to characterizing atomic contacts is the Smooth Particle Method pioneered by Betz and coworkers.[26] In this approach, through convolution against a kernel function, the non-dimensional atomic particles are given volume. Contact can then be explicitly defined as the intersection of the isosurfaces generated by this convolution routine. A particularly appealing feature of this technique is that it only depends on atomic positions, not the force fields employed, rendering it quite versatile. Because spatial binning of the atomic positions is not required, this measurement method is also particularly suitable for small systems where thermal motion of single atoms could have an overstated impact on the measured distribution of forces and strains. This method has been employed to investigate the contact between asperities in the boundary lubrication regime[26,28] and to determine the real contact area between surfactant monolayers on surfaces.[280] Unfortunately, because the forces between atoms are neglected in modeling the contact, it is insensitive to the distribution of pressure at the interface, which must be obtained through other means. The ability to characterize pressure transmission and redistribution through a boundary film could be profoundly beneficial in their development, however, and the methods developed herein seek to

achieve this. Furthermore, the ability to characterize total area of interaction, true contact area between substrates, and pressure transmission and distribution in molecular films can be applied to AFM adhesion measurements where detailed understanding of the tip-surface interactions is critical for the quantitative determination of molecular forces and local surface mechanics.[39,40,281]

In this work, we have developed methods for examining the transmission and distribution of pressure through boundary films and coatings at asperity contacts via atomistic simulation. These methods were employed to consider the role of surface packing density and surface morphology on the properties of an OTS film bound to silica nanoparticles, which are employed as model asperity surfaces. OTS ($CH_3(CH_2)_{17}SiCl_3$), a simple 18-carbon chain bound to the surface via covalent siloxane bonds and hydrogen bonding interactions, is a commonly employed molecule to modify the surface energy and surface forces of metal and metal oxide surfaces. In future work, we will use these methods to examine the specific adhesive and frictional characteristic of OTS films and the underlying mechanisms, while here we will demonstrate the application of this method to general parameters relevant to any surface coating, including film packing density and asperity contact geometry. Similar to the Smooth Particle Method, the pointwise atomic forces and positions are convoluted against a kernel function to broaden the atomic forces into two-dimensional pressure maps. By decomposing the atomic forces, pressure imposed at different interfaces can be determined separately, so that the pressure exerted on and transmitted through the adsorbed film can be distinguished. By fitting the pressure maps to continuum models, it is further possible to explore the effects of the coating and

the evolution of the total contact area, and the residual contact area between the underlying substrates, as a function of film density and contact morphology. Though these methods are incapable of accurately modeling the chemistry at the interface, by isolating the mechanical components of contact in the presence of an adsorbed film, it is possible to more clearly understand the friction response of boundary lubricated systems and more effectively optimize the friction response and wear resistance of surface coatings and boundary lubricants, making this approach broadly applicable.

4.2 Methods

4.2.1 Computational Methods

All simulations were performed with the LAMMPS software package developed at Sandia National Labs[88] on Texas A&M University's *Eos* and University of Texas' *Lonestar* high performance computing clusters. Substrates and films were generated as described previously[282] and post-analysis was performed with custom-made scripts and programs. Rendering was performed by the Visual Molecular Dynamics viewer. The Scanning Probe Image Processing Software (Image Metrology, Denmark) was used to visualize the pressure distribution maps.

4.2.2 Simulation Setup

Asperity interactions are modeled by placing two equilibrated silica surface structures a fixed distance apart. One surface was held stationary, while the other was translated or allowed to move freely in the axial direction in response to applied loads. Contact simulations were conducted for asperity-flat and asperity-asperity configurations. Asperity-asperity interactions represent the contacts with greatest pressure in a

macroscopic contact, while the asperity-flat interactions allow for further examination of morphology effects as well as offering insight into AFM tip interactions with SAMs. Preparation of the silica nanoparticles and functionalization with OTS has been previously described.[282] Asperities were modeled as 7 nm diameter silica nanoparticle shells, with a shell thickness of 1.5 nm. Silica disks were modeled with the same thickness and a diameter of 15 nm. The contact simulation models are shown in Figure 4.1, with rigid regions used for applying and maintaining the compressive load indicated. Contact simulations were conducted on unfunctionalized contacts, contacts in which one side was functionalized with OTS, and in which both sides were functionalized. The OTS was covalently bound to the surface by at least one siloxane bond, with proximity conditions used to add additional surface bonds, cross-linking bonds, and hydroxyl groups. Packing density of the silane films considered were 1.5, 2.25, 3.00, and 3.75 molecules/nm^2. All

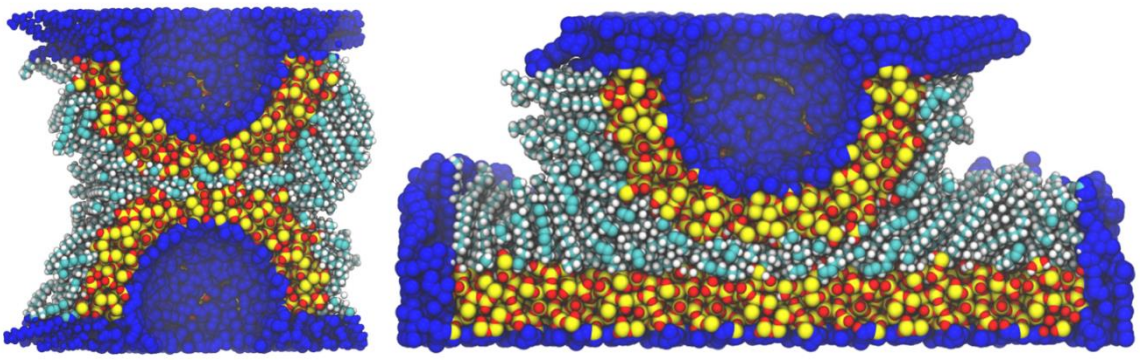


Figure 4.1. Models of the asperity-asperity and asperity-flat contact configurations shown, with atoms held rigid shown in blue. The rigid regions are used to maintain the structure of the nanoparticle, to apply and maintain compressive load, and to shield the contact region from the simulation box boundaries.

contacts were sampled in triplicate by varying the orientation of the particles and position of the flat surface discs.

The OPLS-AA forcefield[118] with additional terms for modeling the silica surface[283] was used to model interatomic interactions. The "SHAKE" algorithm[94] incorporated in the LAMMPS package was used to facilitate larger time steps by iteratively minimizing the C-H/O-H bond lengths and H-C-H angle bending interactions, effectively removing the fast-timescale motion of the hydrogen atoms. The simulations employed a time step of 2 fs, and a Langevin thermostat was applied to all integrated atoms in all dimensions to maintain a temperature of 298 K.

To apply translations and forces to the moving particle, and to maintain the overall structure of the solid surfaces, the particles and flat surface disks each contained a rigid section of atoms behind the freely moving shell. The surfaces were initially placed out of contact, then one particle was translated towards the opposing surface at a velocity of 10 m/s, until slightly repulsive contact was achieved. A 100 nN load was then applied to the moving particle, and the system equilibrated for 3 ns. The contact model was then held fixed and equilibrated for 3 ns, and measurements were collected over 500 ps. To sample the force-distance response, the moving particle was translated out of contact in 2 Å increments, with similar 3 ns and 500 ps equilibration and measurement periods, respectively. Reference simulations were also conducted in which interactions between the surfaces were ignored.

4.2.3 Simulation Measurements

Net forces across the contact were measured in terms of total interaction force, by summing the forces of atoms imposed by one group of atoms upon another:

$$\overrightarrow{F_{J\to I}}(d) = \left\langle \sum_{i}^{I} \sum_{j}^{J} \overrightarrow{F_{j\to i}} \right\rangle_{NVT} \quad \text{..... (4.1)}$$

Here, the interaction force between a group of atoms J imposed on another group of atoms, I, is determined. The forces considered here include van der Waals interactions, and harmonic bond stretching and angle bending interactions present where the film is chemically bound to the surface. While the net force in the contacts can be measured by extracting the forces on the rigid sections that maintain the contact, only the total normal force can be determined in this manner. Using the interaction force in Equation 4.1, it is possible to determine the forces at each of the interfaces in the simulation, in particular the solid-solid and solid-film interface. Interaction forces were collected at 50 fs intervals during the 500 ps measurement period.

4.2.4 Contact Modeling

To spatially analyze the contacts, per-atom interaction force and strain measurements were collected. Due to the long correlation time of per-atom measurements, and in order to reduce the effects of thermal noise, an approach adopted by Harrison and coworkers[284] was used to measure per atom quantities, wherein the property A of atom i, A_i, is determined by the following relation:

$$A_i = \frac{1}{N} \sum_{\tau}^{N} A_i(\tau) \quad \text{..... (4.2)}$$

Atomically detailed information was sampled every 50 fs, with average values reported every 10 ps, resulting in 50 atomic datasets generated over the course of the 500 ps measurement period.

To model the pressure distribution of the contacts, the atomic point forces were "smeared" onto the 2-dimensional contact plane. A convolution scheme was used to smooth the pointwise atomic position and forces across the contact plane:

$$p(x,y) = \int\left(\sum_i F_{z_i}\delta_{x`-x_i}\delta_{y`-y_i}G_i\left(x-x`,y-y`\right)\right)dx`dy` \quad \text{..... (4.3)}$$

This approach is similar to the SPM,[26] however smoothing is only conducted in two dimensions and the kernel function:

$$G_i(x,y) = \left(\sigma_i\pi\ln(2)\right)^{-1} Exp\left(\frac{-\left(x^2-y^2\right)}{\sigma_i\ln(2)}\right) \quad \text{..... (4.4)}$$

was a non-compact Gaussian function, defined such that the full-width at half maximum corresponds to the atomic van der Waals radius (σ_i). The SPM employs a similar, though compact, cubic spline kernel, however the lower dimensionality of this method prevents the non-compact Gaussian kernel from being prohibitively expensive. Each atomic dataset was used to construct a pressure distribution map, and the results averaged to capture the variability in the atomic positions across the contact plane. Pressure profiles and maps presented in this article are further averaged over the three contact orientations. Examples of pressure maps obtained from an OTS-functionalized contact are shown in Figure 4.2, with the total interaction pressure and only the substrate contact pressure maps are shown, as well as the corresponding interaction radius.

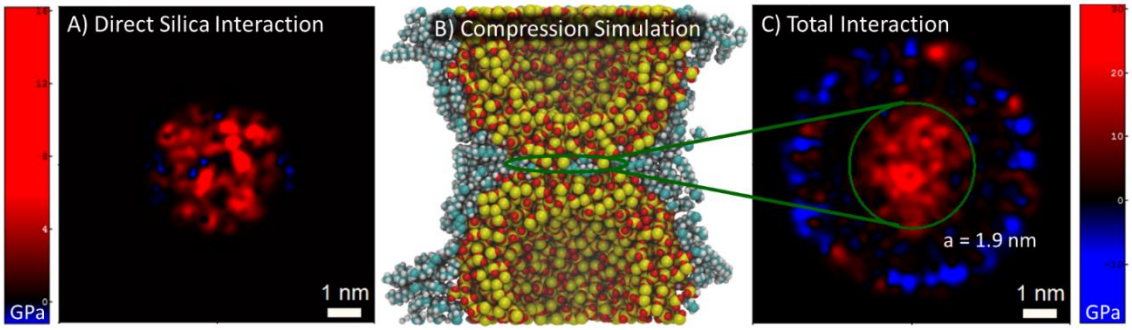


Figure 4.2. Asperity-asperity contact of two OTS-functionalized silica asperities in compressive contact (B) with the total interaction pressure (C) and the direct silica interaction pressure (A).

The resulting pressure maps were then fit to the Hertz continuum contact model.[12] Fitting was achieved with a thermal annealing routine,[285] in which the "energy" of the system was defined as:

$$E(\vec{\nu}) = \int \left(p_{\text{model}}(x, y; \vec{\nu}) - p_{\text{data}}(x, y) \right)^2 dxdy \quad \text{..... (4.5)}$$

Where $\vec{\nu}$ represents the parameter space of the model. For the Hertz contact model used in this work, the pressure distribution is given by:

$$p(r; p_0, a) = p_0 \left(1 - {}^{r^2}\!/_{a^2}\right)^{1/2} \quad \text{..... (4.6)}$$

Where the parameter space consists of only the contact radius, a, and the peak pressure, p_0, though this approach is easily expanded to consider other contact models like the Johnson-Kendall-Roberts model for soft contacts with adhesion. The hardness of the silica nanoasperities examined here dictates the application of the Hertz contact model, though within the contact radius, the pressure distribution is identical to that of the DMT model for contact between hard surfaces with adhesion. Fitting was performed on pressure

maps for individual orientations, and the average results are reported.

4.3 Results and Discussion

4.3.1 Dry Contact Modeling

To demonstrate and validate the results of the models, and as a baseline for determining the impact of the OTS film, the contact properties of bare asperity-asperity and asperity-flat interactions were examined. The force-distance response is shown in Figure 4.3, with corresponding contact stiffness reported in terms of the reduced elastic modulus. The contacts were found to be substantially harder than would be expected

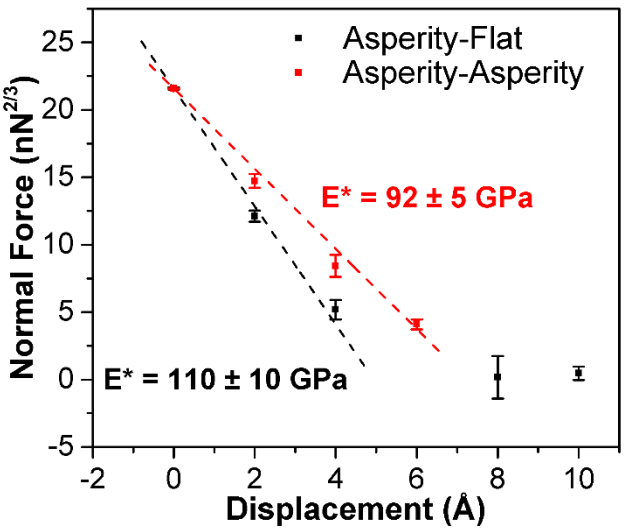


Figure 4.3. Force-distance relationship for unfunctionalized surfaces in the asperity-flat(black) and asperity-asperity(red) configurations. The corresponding reduced elastic moduli for the contacts are shown ($F^{2/3}=((16/9)rE^{*2})^{1/3}d$), determined from the observed forces in repulsive contact, demonstrating reasonable agreement in the mechanical properties of the two substrates. The modest discrepancy in the values is likely due to greater sampling of the rigid backing of the flat surface, giving rise to a more stiff contact. Slightly adhesive points do not survive the transformation and are not shown.

for a real silica contact (*ca.* 53 GPa), likely owing to the rigid backing of the substrate surfaces and static bonding between the atoms that prevents more realistic deformation of due to greater effect from the rigid backing of the flat surface. Taken over the entire area of the contact, the average distance between the area of contact and this rigid plane is smaller for the asperity-flat contact.

The pressure distribution for these two contacts is depicted in Figure 4.4, for contact at 100 nN applied load, with corresponding best fit Hertz pressure profiles shown. Excellent agreement in the contact region is obtained for the asperity-flat interaction, though the contact radius of 1.8±0.1 nm was significantly less than the radius predicted by Hertz theory of 2.4±0.2 nm. Conversely, the asperity-asperity pressure profile disagreed with the Hertz model, particularly at the center of the contact, but the contact radius of 1.4±0.1 nm was in good agreement with the Hertz prediction of 1.4±0.1 nm.

Disagreement between the pressure profile and Hertz model was not observed at lower pressures, and was attributed to the rigid core of the particle influencing the mechanical behavior of the contact at extreme pressure. In both cases, there was disagreement at the edge of the contact area, with the contact pressure tailing off more gradually than predicted by the Hertz contact model. This has been previously observed in atomistic simulation[182,279] and represents a deficiency of the continuum model at atomic length scales due to the non-finite nature of atomic interactions. It is important to note that the relatively regular pressure distribution in these profiles hides considerable inhomogeneity the surfaces. The modest difference in stiffness between the two morphologies is likely

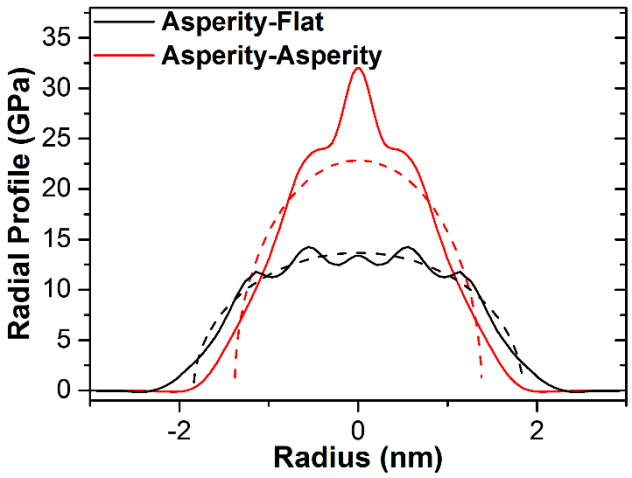


Figure 4.4. Pressure profiles for the bare asperity-asperity and asperity-flat contacts at 100 nN applied load. The dotted lines depict the best fit to the Hertz contact model. The asperity-flat contact provided reasonable agreement with the best fit Hertz model, while the asperity-asperity interaction demonstrated deviation at the center of the contact like due to the rigid backing of the particle.

in the 2-dimensional pressure distribution across the contact plane, visible in Figure 4.2. This irregularity in the pressure distribution is a result of roughness that exists at the smallest length scales.

Examination of the bare contacts exposed some of the weaknesses of the model system, which are inherent to the force fields used and the geometry and structure of the solid substrates. The forcefield is simply not optimized for modeling the mechanical behaviors of the silica, nor is a fixed binding topology appropriate for contacts at this magnitude of pressure, resulting in more rigid contacts. While this will result in quantitative inaccuracy, the results can be used as a baseline for considering the qualitative effects of film introduction and as a basis for comparison and development of surface coatings. Though more advanced force fields like bond-order potentials may have

improved the quantitative accuracy of the contact simulations, the many-body interactions in these force fields renders the distinction of forces between two atoms difficult if not impossible to define. More simplistic force fields consisting of only 2-body Lennard-Jones or Morse potential interactions are better able to reproduce plastic deformation and have been used in modeling solid contacts,[11] however such an approach could limit the complexity of the adsorbed film. While the choice of a molecular forcefield does negatively impact the mechanical behavior of the solid substrates, it will better model the behavior of the film which is of greater interest in this work. Fortunately, as the film is introduced into the contact, the magnitude of the pressures and strains imposed on the solid substrates is reduced, and these effects vanish rapidly as a result.

4.3.2 Film Effects on Asperity Contact Mechanics and Pressure Distribution

With introduction of OTS to the contact, the pressure distributions were found to broaden, increasing the total interaction areas and reducing the peak pressures at the interface. Relatively linear relationships between film density and Hertz contact parameters of contact radius and peak pressure were observed, depicted in Figure 4.5. Pressure profiles corresponding to increasing numbers of molecules bound within the contact are shown in Figure 4.6. At the lowest coverage examined, deviation from the Hertz contact model is still observed at the center of the contact, due to the extremely high pressure in this contact. This feature vanished with increasing coverage as the maximal pressure in the contact was reduced, however it reappeared at the maximal coverage examined. The latter behavior is likely due to decreased deformation of the silica as the film softens the contacts, indicating reduced compressive strain of the solid surfaces.

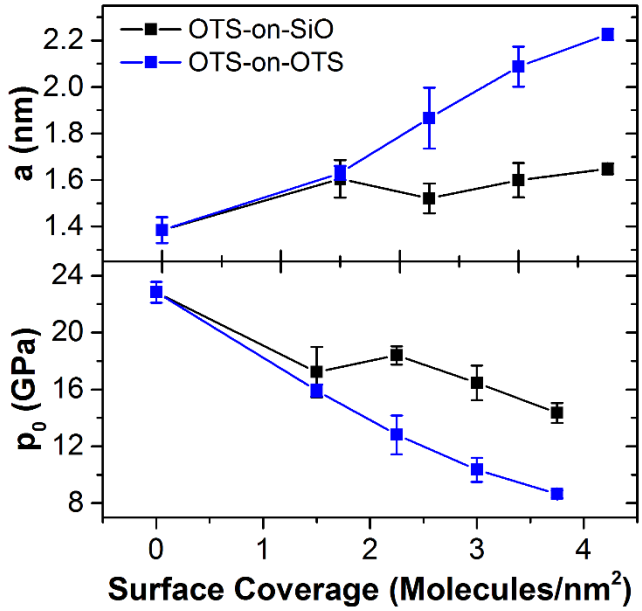


Figure 4.5. Contact radius (a) and peak pressure (p*0*) determined by fitting pressure profiles of asperity-asperity contacts with increasing surface coverage of OTS. Relatively linear relationships between film density and these contact properties are observed, with contact radius increasing and peak pressure decreasing as film density increases. Lines are provided only to guide the eye, and where no error bar is visible, the error was smaller than the size of the symbol.

Additionally, the silica interaction became more diffuse with introduction of the film, though when both sides were functionalized with OTS, the silica interactions tended to increase due to increased density of siloxane linkers on the surface that were treated as part of the substrate.

The role of direct substrate interaction is of critical importance. Metal and oxide surfaces are generally high energy interfaces with many available tribochemical reaction pathways. The adsorbed film can terminate these pathways by minimizing contact between the interfaces. The lowest coverage case shown in Figure 4.6A, for example,

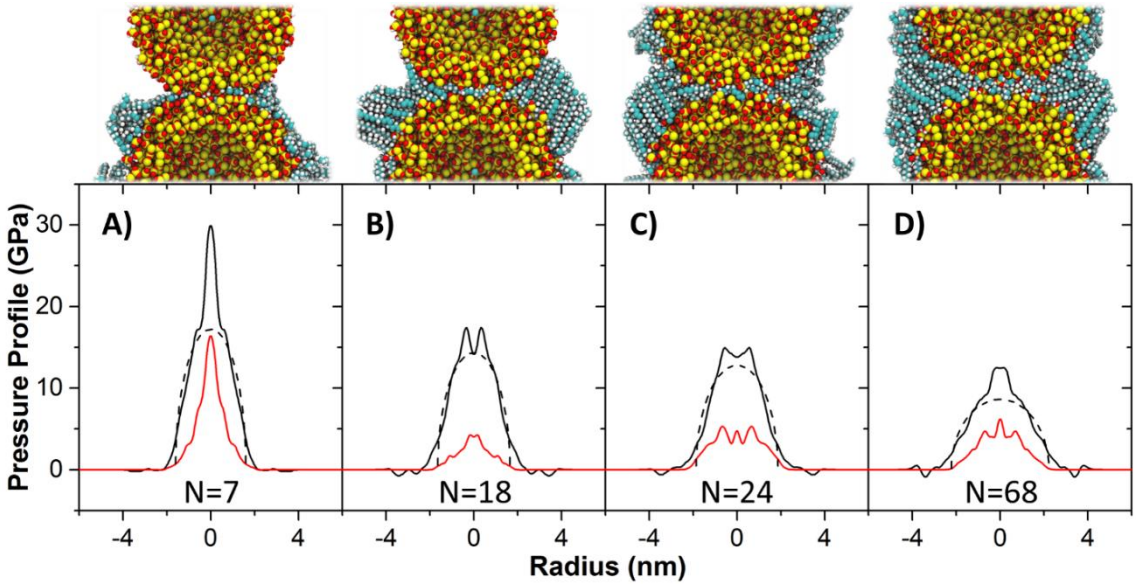


Figure 4.6. Pressure profiles as a function of film coverage for asperity-asperity contacts functionalized with OTS. Total contact pressure (solid black), best fit Hertz contact profile (dotted black), and silica-silica contact pressure (red) are shown. With each set of curves, the number of molecules bound within the contact (N) is indicated. The profiles correspond to: A) 1.5 molecules/nm^2 on one side; B) 3.75 on one side; C) 2.25 on both sides; D) 3.75 on both sides.

demonstrates substantial interaction between the underlying interfaces that would likely lead to substantial wear during sliding and a greater likelihood of film depletion. In order to characterize the evolution of the substrate contact, the silica contacts were analyzed using the same methodology, obtaining the corresponding contact radii and maximum pressure for the direct substrate interaction. Considered only in terms of surface coverage, it is difficult to identify a clear trend in the evolution of the substrate contact properties with film density, but as a function of the number of molecules within the contact, determined by projecting the total contact area onto the surface or surfaces of the particles and scaling by the coverage density, consistent behavior was observed. The upper pane

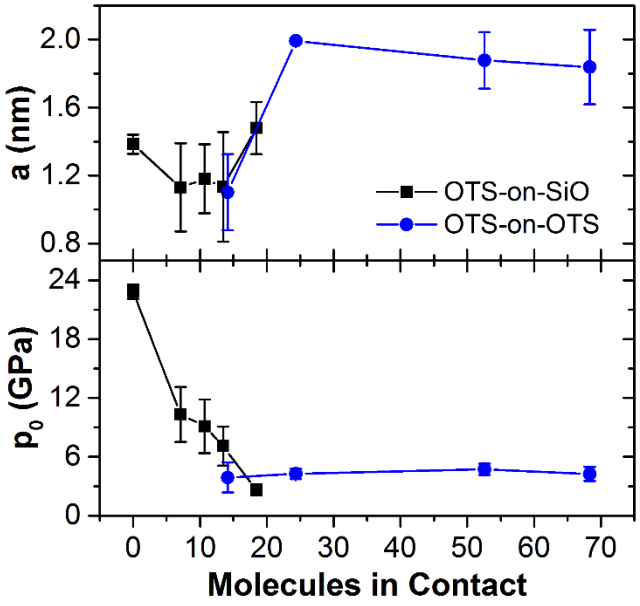


Figure 4.7. Contact radius (a) and peak contact pressure (p*0*) between the silica substrates for asperity-asperity interactions as a function of the number of molecules bound to the surface or surfaces within the total interaction area. Upon initial introduction of the film, the silica contact radius decreases and remains constant. At a threshold coverage, the contact radius substantially increases, and again remains constant. These behavioral regimes correspond to a steady decline in the peak contact pressure and stabilization of the contact pressure, respectively. Where no error bar is visible, the error was smaller than the size of the symbol.

of Figure 4.7 depicts the evolution of the substrate contact radius as a function of coverage density, wherein two behavioral regimes are observed. At low density, the contact radius remains relatively constant near 1 nm, and at a certain threshold density, this value increases substantially to approximately 2 nm, corresponding to a 4-fold increase in the contact area. The peak contact pressure at the silica-silica interface, shown in the lower pane of Figure 4.7, similarly demonstrates two regimes, a steadily declining peak pressure corresponding to the first regime, and a relatively constant peak pressure corresponding

to the second. This behavior indicates a sharp transition from a dry to lubricated contact, analogous to the dry sliding and boundary lubrication regimes, respectively. The exact nature of this transition point in real contacts likely depends on a variety of factors, including applied pressure, surface curvature, and contact stiffness. The density of lubricants at the interface is also determined by a variety of factors, including affinity or chemistry of the interfaces, surface morphology, local hydrodynamic pressure at the contact, and the physical and chemical structure of the lubricant.

4.3.3 Surface Morphology and Film Effects

To better understand how these factors depend on surface morphology, asperity-flat interactions were also examined as a function of film density. Not only does this configuration change the curvature of the contact, it introduces asymmetry that can be used to examine the differences in behavior of films identical in terms of coverage on the two different surfaces of the contact. Figure 4.8 depicts the evolution of the contact profiles for contacts in which the flat surface had been functionalized. At the lowest coverage density, some irregularity of the pressure within the contact was observed, however the deviations from the fit are offsetting. With increasing coverage, however, a peak in the pressure profile at the center of the contact was again observed, corresponding to decreased deformation of the asperity. Additionally, with increasing film density, the substrate interactions in the contact became vanishingly small, with the vast majority of the contact load supported by the silica-film interface. OTS SAMs on flat surfaces have been observed by AFM to be quite robust,[38] and this is likely because the film effectively terminates the tribochemical pathways of the silica interface by inhibiting interactions

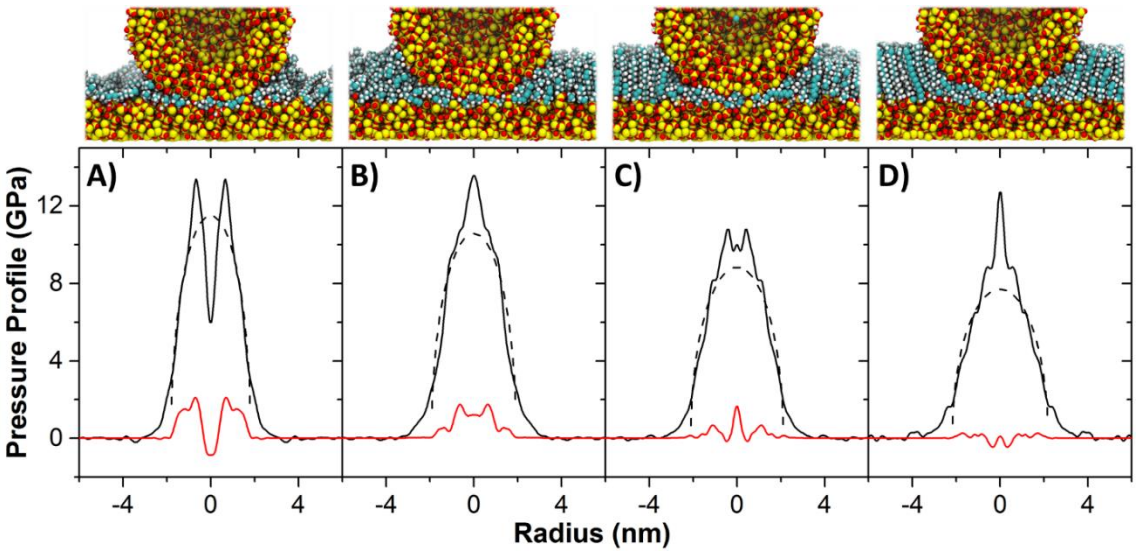


Figure 4.8. Pressure profiles of asperity-flat interactions in which the flat surface is functionalized with OTS of increasing packing density. The evolution in total contact pressure (solid black), best fit Hertz pressure profile (dotted black), and silica-silica interaction pressure (red) are shown. As the film packing density was increased, deviation from the Hertz contact model was observed at the center of the contact, suggesting reduced deformation of the solid surfaces. The direct silica-silica interaction also becomes vanishing small, reducing to long range adhesive interactions, at maximal film packing density. Film density in the contacts are A) 1.5 molecules/nm^2; B) 2.25; C) 3.00; D) 3.75.

between the two surfaces.

The magnitude of the direct silica contact was examined as a function of the number of molecules in the contact, as shown in Figure 4.9. The contact radius of the direct silica interaction could not be accurately determined because the silica pressure was very small, however fitting of the peak pressure was still feasible and produced relatively reliable results. For the two configurations, the number of molecules bound within the contact area was greater when the film was applied to the asperity, however molecules outside the area of contact would be expected to have greater influence on the flat surface. The

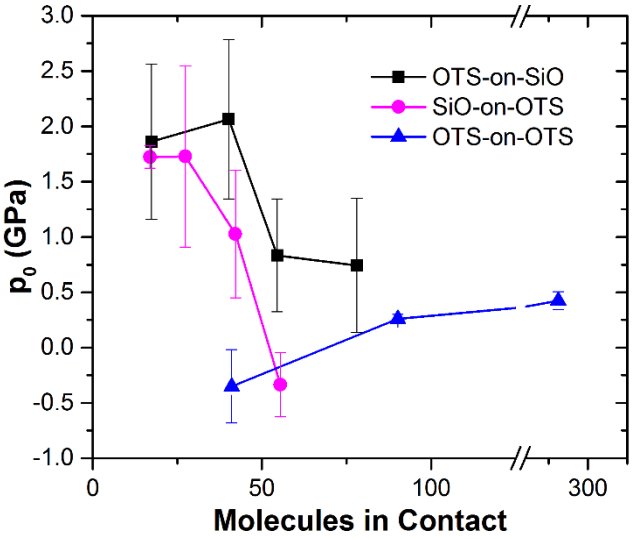


Figure 4.9. The peak pressure of the silica-silica interaction profiles as a function of the number of molecules in the contact. Though the surface morphology is held constant, the behaviors observed distinctly depend on whether the film is adsorbed to the asperity, flat surface, or both. In particular, the mitigation of surface contact for the film on the asperity surface is substantially less than that of the film on the flat surface, even for substantially numbers of molecules bound within the contact. The increase in contact pressure observed when both sides are functionalized, similarly observed for asperity-asperity contacts, is likely a result of increased silane-silane interactions resulting from both sides of the contact being functionalized.

peak contact pressure at the silica interface was uniformly lower when the film was applied to the flat surface, indicating that a chemically equivalent monolayer on a low curvature surface is better able to minimize contact than a monolayer on a high curvature contact, which is consistent with AFM investigations of OTS films.[176]

Comparing the differences in more detail, Figure 4.10A depicts the pressure profiles for contacts with identical surface binding chemistry (i.e. packing density of 2.25 Molecules/nm^2) at the interface. Interestingly, the total pressure profiles were very

similar, however the peak silica interaction pressure was observed to nearly double when the film was bound to the asperity. Figure 4.10B similarly depicts the pressure profiles when similar numbers of molecules are bound within the contact, in this case approximately 40 molecules. Here, the flat surface was observed to have both lower overall pressure, and substantially reduced silica interaction. While variations in the frictional[281] and adhesive[176] response of OTS SAMs has been observed in AFM microscopy experiments, depending on whether the AFM tip or the surface is functionalized, these results cannot be entirely attributed to morphological differences, as the packing density of OTS SAMs on asperities and flat surfaces has been observed to be

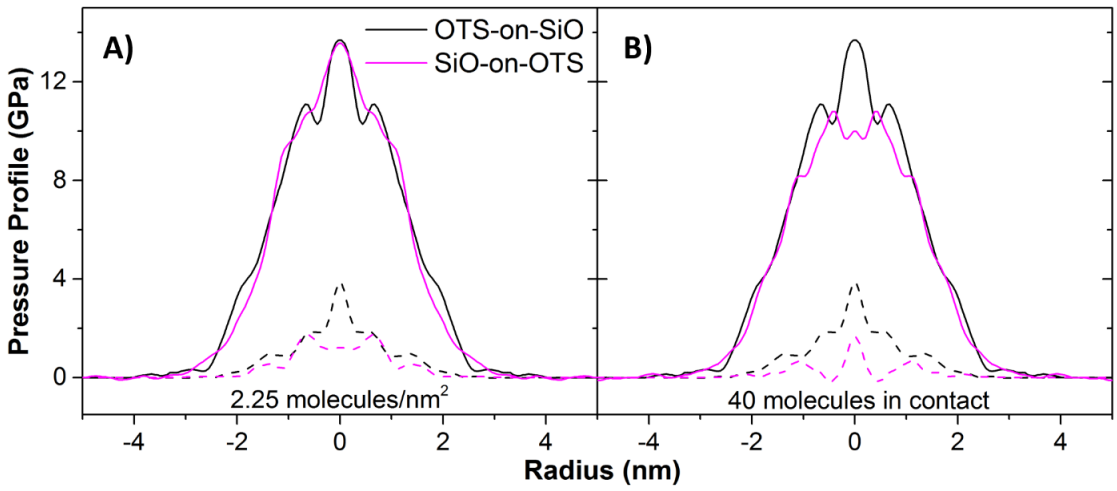


Figure 4.10. Pressure profiles of OTS films in asperity-flat interactions with identical packing density of 2.25 Molecules/nm^2 (A) and nearly identical numbers of molecules, approximately 40, bound within the contact (B). In the former, nearly identical total pressure distributions are observed, though the silica contact is nearly double when the film is adsorbed to the asperity. In the latter, lower overall and direct silica pressure is observed when the film is adsorbed to the flat surface.

strikingly different.[32,33,216] The differences in surface protection of otherwise chemically identical films however likely plays a role in the increased friction and adhesive response of OTS films on asperities.

4.4 Conclusion

Methods were developed to examine and fit atomistic surface contact simulation to continuum models in the presence of an adsorbed film. To demonstrate the applicability of this method, the behavior of an OTS film in silica asperity-asperity and asperity-flat contacts was examined as a function of film density and geometry of the contact. With increasing density, a relatively linear rise in the contact radius, and corresponding decline in the peak contact pressure were observed. Additionally, deviations from continuum model pressure distributions arose as a consequence of decreased deformation of the asperity surfaces as the contacts were softened with introduction of the film. Redirection of pressure away from the silica-silica interface was also observed, though this effect was much greater for the contact with greater reduced radius of curvature. The methods developed here can be employed to consider a variety of molecular coatings and boundary lubricants to determine and optimize their behavior at asperity contacts. Furthermore these results provide insight into the interaction of AFM tips with modified surfaces, including the surface forces of adhesion and friction and wear of surfaces. In future work, these methods will be applied to specifically provide a better understanding of friction and adhesion measurements of OTS films in experiment and applications.

CHAPTER V

THE ROLE OF SUBSTRATE INTERACTIONS IN THE MODIFICATION OF SURFACE FORCES BY SELF-ASSEMBLED MONOLAYERS[*]

5.1 Introduction

Over the past 30 years, the self-assembled monolayer has become a mainstay of surface modification.[244,286,287] Their simple chemical structure, ease of application, and high tailorability allow researchers and developers to modify the chemistry of interfaces in well-controlled fashion. An area of relatively recent interest has been the modification of surfaces with SAMs to achieve improved friction response and wear resistance, particularly in applications in MEMS,[36,141,288] where traditional lubrication is not feasible, surface coatings are one of few viable alternatives, and contact forces are relatively small. Traditional, covalently bound SAMs proved relatively ineffective in preventing wear[146,289] even at these very small sliding interfaces, but self-healing approaches like vapour phase lubrication with SAMs[290] and even simple alcohols[25] have met with some success. Furthermore, SAMs provide an excellent platform for investigating dissipative mechanisms at interfaces with adsorbed monolayers as their structure and chemistry is well defined.

The ability of SAMs to modify the surface forces that dictate friction and adhesion at

[*] Reproduced from Ewers, B. W.; Batteas, J. D. "The role of substrate interactions in the modification of surface forces by self-assembled monolayers". *RSC Adv*. **2014**, 4, 16803-16812, with permission from The Royal Society of Chemistry.

interfaces has been well documented.[291,292] To investigate fundamental processes in the dissipation of energy at sliding interfaces, precise control of the geometry and chemistry of the interfaces is required, therefore friction response is best measured using methods where the contact area is either measurable or at least predictable. This is achievable with the SFA,[293] the IFM,[294] or the AFM.[123] In a vast majority of cases the friction response of sliding interfaces in these experiments is consistent with the laws of single asperity friction,[23,256] that is, the friction force is proportional to the contact area:

$$F = \tau A \quad \text{..... (5.1)}$$

Where τ is the interfacial shear strength, a measure of the lateral stiffness of the contact. The friction response of SAMs has in many cases been observed to be consistent with the single asperity friction law.[295] Many other cases exist of single asperity friction responses consistent with Amonton's law,[163,296-298] wherein friction response is directly proportional to the contact load with no apparent dependence on contact area. Leggett and co-workers have extensively examined the friction response under controlled solvent environments, observing friction consistent with both Amonton's law and single asperity friction laws, depending on the extent of solvation of the SAM interface.[39,41] They find that the friction response of SAMs can effectively be understood in terms of a three term friction law[27] that is essentially a combination of the single asperity friction law and Amonton's law:

$$F = \tau A + \mu L + F_0 \quad \text{..... (5.2)}$$

In addition to the interfacial shear strength, μ is the traditional load dependent term referred to herein as simply the friction coefficient. This term perhaps arises from chemical interactions at the interface, as it has been shown that the number of atomic

contacts is proportional to the applied load for non-adhesive contacts.[181,299] The Derjaguin offset, F_0, has been suggested to be related to the zero-load contact area of the molecular monolayers.[280] Because the cohesive forces of SAMs consist of only weak van der Waals interactions, the shear strain is of limited magnitude and spatial extent, thus minimizing shear related dissipation and thereby limiting the interfacial shear strength.[40,105] A low interfacial shear strength results in a friction response that is dominated by the load dependent term, and a response consistent with Amonton's law. Alternatively, because the SAM effectively decouples the two sliding interfaces, the contact area may be viewed as only the contact area between the asperity surfaces,[27] which for sufficiently low loads may be negligible. This can potentially be manipulated by increasing the adhesion at the interface, which can be achieved by using solvents that do not sufficiently solvate the end-groups of the SAM. The increased molecular interaction at the SAM interface thereby leads to greater shear strength in the contact due to increased coherence of the SAM molecules during sliding or increased mechanical coupling of the interface.

Alternative explanations for this behaviour have been proposed by Szlufarska *et al.*,[177] observing by molecular dynamics simulation of a hard asperity contact that, for a non-adhesive contact, friction and true contact area are proportional to load. This result was obtained by completely neglecting long range van der Waals interactions. Because SAM molecules lack the short range structural rigidity of a solid substrate, even these weak forces will play a significant role in the extent of atomic contact, so though perhaps valid for hard surface contacts, their result is likely not applicable to friction on SAM coatings.

Where the friction of SAMs becomes particularly unclear is at greater applied

pressures. Salmeron *et al*. identified four different friction regimes for OTS derived SAMs on mica examined by AFM,[38] ranging from elastic dissipation mechanisms at low load to response dominated by wear of the substrate at the greatest loads. Carpick *et al.* examined the friction response for OTS SAMs by AFM, examining the role of SAM configuration within the contact.[281] They observed single asperity friction response when the SAM was applied to the AFM tip, but Amonton's law and higher order behaviour at higher loads, when the SAM was applied to the opposing surface or to both surfaces. Higher order behaviours are not predicted by the simple three term law, which ranges from sub-linear to linear with respect to applied load. Moreover, because these films are very thin and quite compliant, it is ultimately necessary to consider what role the substrate plays in the friction response. Increased mechanical coupling between the bulk substrates would be expected to result in increased interfacial shear strength, and chemical interactions between the more reactive surface substrates would result in a greater coefficient of friction.

Unfortunately, examining the contact in such a way that segregates substrate and film interactions is not achievable experimentally. From friction data it is possible to determine contact area, but only when one can safely assume that friction is directly proportional to area and the mechanical properties of the interface are well known.[18] Mechanical characterization of SAMs is lacking and likely very dependent on the substrate and film preparation conditions, and the dominance of the load dependent term in many cases undermines the essential assumption of proportionality for analysis by the general equation. Even for cases in which sub-linearity is observed, fitting to the general equation

may simply mask linear contributions to the friction response.

MD simulation is an alternative which, though unable to completely characterize the atomistic behaviour of an asperity contact, can be used to provide useful insight into some of the mechanical and conformational dynamics of asperity contacts at an atomistic level. MD simulations have long been used to characterize contacts and friction behaviour for flat and asperity contacts[187] with[208,209,274] and without[278] surface adsorbates, and a variety of methods have been put forth to characterize contact area and pressure distribution in these contacts. We have developed methods for analysing the total area of interaction and characterization of the true substrate contact for asperity-flat and asperity-asperity contacts coated with self-assembled monolayers.[300] These determinations are based on segregated analyses of the film-substrate and substrate-substrate interactions at the interface. These techniques have been used to explore the relationship of film packing density and morphology to the total interaction area and true contact area in these contacts. Herein, the results of these methods were used to explore measurements of surface forces in SAM protected contacts to better understand the role of substrate interactions and alternative dissipation mechanisms in the adhesive and frictional forces observed.

5.2 Contact Simulation and Characterization

Atomistic simulations of asperity contacts were conducted using OTS-functionalized silica nanoparticle and flat substrates developed previously.[282] The asperity surfaces employed in this work have a radius of curvature of 3.5 nm, and the film packing densities were varied from 1.5 to 3.75 molecules/nm^2 for the covalently bound film. Covalent functionalization was chosen, as opposed to a primarily physi-sorbed, or hydrogen bonded

film because it has been found that these structures are not likely to exist on surfaces with high curvature.[32,33,250] The molecules were attached to the surface by at least one siloxane bond, and proximity conditions were used to assign additional bonds to the surface, crosslinking bonds to other silanes, or the addition of hydroxyl groups. The contact simulations are designed to simulate traditional AFM experiments, in which an AFM tip is in contact with a flat surface, as well as asperity-asperity interactions achieved in AFM by coating the opposing surface with a nanoparticle film. A representation of the experimental configurations and the corresponding simulated geometries are depicted in Figure 5.1.

The modified all-atom OPLS forcefield[118] with additional terms for the silica surface were employed,[283] and the SHAKE algorithm[94] was used to constrain the fastest timescale

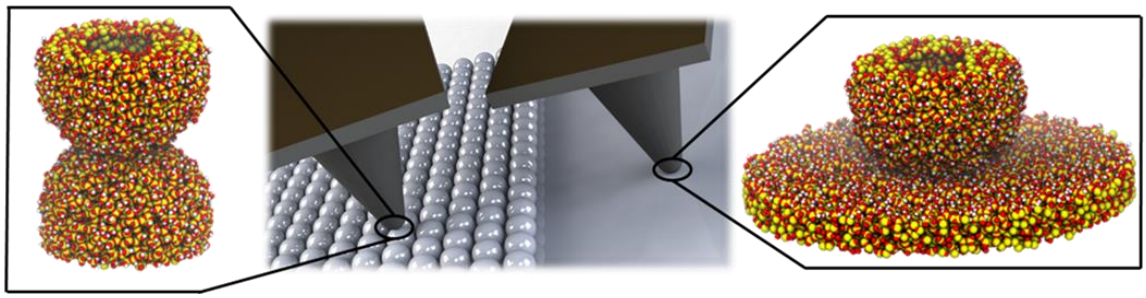


Figure 5.1. The simulation methodology is designed to mimic surface nanoasperity interactions. These can be made in the laboratory by AFM, wherein a nanoscopically sharp AFM tip is brought into contact with a surface. The model for typical AFM experiments is shown to the right, wherein a nanoscale asperity is brought into contact with a flat surface. To mimic asperity-asperity interactions as would be observed at the contacts of real surfaces, nanoparticles may be employed to impart tuneable local surface curvature to the opposing surface, the simulation model for this type of contact is shown on the left.

motion of the hydrogen atoms, providing greater computational efficiency while still providing for fully atomistic simulation of the hydrocarbon chains. While many of the discussions that follow will consider the propensity for wear to occur, we emphasize that this choice of force field prevents the actual formation and scission of bonds at the interface, so the simulation results may only be used to identify points where wear is likely to occur. A Langevin thermostat was used to maintain the simulated contacts at 300 K. Integration was performed by the LAMMPS software package developed by Sandia National Laboratories,[88] conducted on the Texas A&M University's *Eos* and University of Texas' *Lonestar* high performance computing clusters.

Contact was achieved by pressing the opposing surfaces into contact at 100 nN compressive load. Rigid portions of the substrates were used to apply a compressive load on one surface, and to hold the other surface in place. The contacts were held under the compressive load for 3 ns, the system was then equilibrated in a fixed position with no compressive load for 2 ns, and measurements were collected for 0.5 ns. Sampling of the force-distance response and evaluation of surface forces at light compressive loads was achieved by pulling the particles apart in 2 Å increments, equilibrating for 3 ns, and performing measurements for 0.5 ns. It is important to note that the simulations are performed while the surfaces are held fixed, and therefore does not represent the dynamic motion of an AFM tip. This is a necessary compromise owing to the fact that the speeds of AFM tip motion are generally much slower than would be accessible to large scale simulations. Because the tip motion is often much slower than the timescale of molecular motion, we have chosen to instead examine the static contacts as opposed to dynamic ones.

To analyse the pressure and strain distributions in the contacts, time averaged measurements of the strains and atomic forces were collected. Samples were collected every 50 fs, with 200 samples collected over 10 ps used to generate individual data sets. 50 datasets were generated during each measurement trial, and each of these datasets was used to generate maps of the various atomic properties in the contact plane. These 50 maps were then averaged together to generate the time averaged property maps. For each set of contact conditions, contacts were simulated in triplicate by using different faces of the particles and different sections of the flat surfaces, and the final property maps from each orientation were averaged and are presented here.

Characterization of the contacts was conducted by convolution of the atomic positions and atomic forces or strain energies, according to Equation 5.3:

$$p(x,y) = \int \left(\sum_i A_i \delta_{x`-x_i} \delta_{y`-y_i} \right) G_i(x-x`, y-y`) dx`dy` \quad \text{..... (5.3)}$$

Where A_i represents the property of atom 'i' mapped into the contact plane, and $G_i(x,y)$ is a Gaussian kernel function defined as:

$$G_i(x,y) = \frac{1}{\sigma_i \pi \ln(2)} Exp\left(\frac{-(x^2+y^2)}{\sigma_i \ln(2)} \right) \quad \text{..... (5.4)}$$

Where the full width at half maximum is the van der Waals radius σ_i of the given atom. These 2-dimensional contact maps are presented as radial profiles by circular integration. Determination of parameters like the contact area and peak pressure are a result of fitting these pressure maps to the Hertzian contact pressure function:[12]

$$p(r) = p_0 \left(1 - r^2/a^2\right)^{1/2} \quad \text{..... (5.5)}$$

Fitting is achieved using a thermal annealing algorithm[285] in which the "energy", defined as the difference between the fit function and the measured pressure map, is minimized over the parameter space 'a', the contact area, and p_0, the peak pressure.

The properties discussed herein include interaction forces and strain energies. Interaction forces are simply defined as the forces imposed by one group of atoms upon another, for example the forces from atoms in one silica asperity directed upon the other asperity. This primarily involves van der Waals interactions except for film-substrate interactions, which include bond-stretching and angle-bending contributions. Strains were similarly segregated to include internal strains and strains imposed by outside groups, so that, for example, the film strain discussed herein is the internal strain of the film, excluding external interactions.

5.3 Results and Discussion

5.3.1 Adhesive Forces at SAM Coated Interfaces

Understanding frictional dissipation mechanisms requires an understanding of the mechanical and chemical forces at the interface. Adhesion force measurements by AFM are particularly attuned to chemical interactions at the interface, providing useful insight into the tribochemical dissipation mechanisms that may be present at sliding interfaces. Both covalent and non-covalent bonding interactions can lead to energy dissipation and increased adhesive force. Furthermore, reconfiguration of the contact interface into a more stable potential well can lead to increased adhesion due to greater barriers to surface separation.

Rate dependent variations in the adhesive forces of SAM functionalized surfaces have

been observed by SFA,[301] IFM,[302] and AFM adhesion experiments,[176] but the nature and timescale of these variations are noteworthy, and likely arise from differences in the contact geometries. Long timescale variations in adhesion were observed by SFA, wherein macroscopic atomically smooth surfaces are brought in contact. Here, the contact pressures are relatively low owing to the large contact area, and the long timescale variations in adhesion were attributed to interdigitation of the molecular films on the opposing surfaces. At higher contact pressures, IFM measurements also show slow timescale variation in adhesive force that has been surmised to arise from slow timescale relaxation of compressed films in the contact.

The simulation methodologies employed here focus on the smallest nanoasperity contacts, best mimicking AFM adhesion measurements, and our attention will focus on understanding the results of a series of AFM nanoadhesion experiments of SAMs on surfaces with different morphologies and SAM configuration.[176] The observations in question are summarized in Figure 5.2, in which it was observed that dynamic variations in the force of adhesion do not occur when an AFM tip is brought into contact with a SAM on a flat silicon surface, regardless of the chemistry of the tip surface. However, when a functionalized tip was brought into contact with a bare silicon surface, or with a functionalized rough silica surface, a clear rate dependence in the adhesive force was observed. This rate dependence was observed to occur on relatively long timescales and we seek to understand the source of these variations by examination of the contacts in atomistic detail. It is important to note that these measurements were conducted in water at pH 3, the isoelectric point of the silica surface, so that Coulombic interactions and

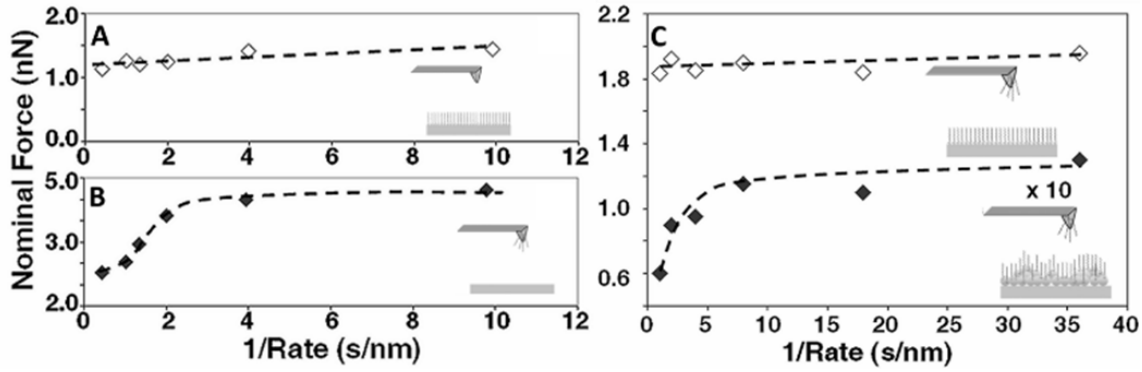


Figure 5.2. Adhesion measurements conducted in various configurations of alkylsilane on flat and silica nanoparticle roughened surfaces of silicon. The specific configuration is depicted with each curve. Dynamic variations in the adhesive force were observed when only the asperity is functionalized, or when a functionalized asperity is brought into contact with a functionalized, but roughened, surface. Adapted with permission from Xu, C.; Jones, R. L.; Batteas, J. D. "Dynamic Variations in Adhesion of Self-Assembled Monolayers on Nanoasperities Probed by Atomic Force Microscopy". *Scanning*. **2008**, 30, 106-117.[176] Copyright 2008 Wiley Periodicals, Inc.

surface chemistry in the aqueous environment were not favourable.

Simulations were conducted that mimic the four configurations examined by AFM at an applied force of 100 nN, shown in Figure 5.3A-D. Figures 5.3A and 5.3B represent contacts for which dynamic variation was not observed, and Figures 5.3C and 5.3D represent contacts for which adhesion was greater for slower approach rates. The defining difference between the contacts which do show dynamic variation in adhesive response, versus those that do not, is a clear repulsive interaction at the silica-silica interface indicative of direct substrate interaction. Chemical processes at these silica-silica interfaces could include the formation of hydrogen bonds between silanol groups on the opposing silica surfaces or the formation of siloxane bonds at the interface, bonds which necessarily must be broken to pull the surfaces out of contact and which can give rise to a

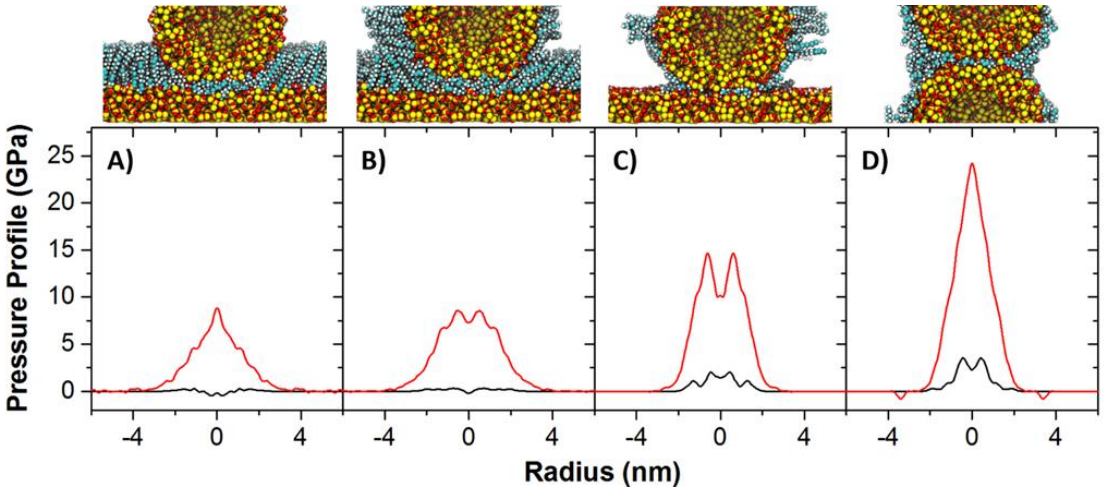


Figure 5.3. Pressure profiles of SAMs in asperity contacts, demonstrating the overall pressure in the contact (black) and the pressure at the silica-silica interface (red). In configurations A and B, which model adhesion measurements in which dynamic variation is not observed, virtually no silica contact pressure is observed, while in configurations C and D, which model adhesion measurements in which variation did occur, there is distinctly greater pressures observed as well as non-negligible pressure at the silica-silica interface that could give rise to pressure induced siloxane bond formation. Packing densities employed here are A) 3.75, B) 2.25, C) and D) 1.5 molecules/nm^2, chosen to best represent realistic conditions.

greater work of adhesion.

A major challenge in interpreting the experimental results is specifically the rate dependent nature of the adhesive response. The variations in the approach/retract rates are over second to millisecond timescales, far slower than typical chemical timescales. Even if the variation in adhesion were driven by the reaction rate of bond formation across the silica interface, the rise in adhesion would be gradual, with a corresponding gradual rise in the adhesive force. Alternatively, the rate could be sufficiently low that the jump in adhesive force is due to a stochastic event, *i.e.* single bond formation, but that would be subject to statistical variation and nevertheless would not give rise to such a large change

in the adhesive interaction. Simply put, it is not reasonable to suggest that, for faster approach rates, there is not enough time for substrate interactions to lead to bond formation.

It is necessary to identify processes that occur on timescales slow enough that would give rise to greater adhesion. Reconfiguration of the compressed SAM at the contact was suggested as a source of greater adhesion in the contacts, consistent with IFM measurements and the notion that the SAMs on these surfaces have some semblance of molecular order. It has since been observed that surface packing density of alkylsilanes on surfaces with nanoscopic curvature is only approximately one third that of a full monolayer,[32,33] and it is for this reason that the simulations shown in Figures 3C and 3D are shown to have extremely sparse OTS coatings (1.5 molecules/nm^2). This sparseness of the film and the nanoscopic curvature of the interfaces collude to eliminate any compressive modes that might play a role in increased work of adhesion, but it also exposes the surface to tribochemical reaction at the silica-silica interface, and these processes can lead to increased adhesion.

To finally reconcile the timescale of molecular orientation, however, it is necessary to identify the dynamical processes that occur on these timescales that can play a role in the silica binding chemistry. It's important to note that the contact simulations are conducted in a vacuum and quickly achieve equilibrium, whereas the adhesion measurements were conducted in an aqueous environment. While the equilibrium state may differ depending on the environment, the very high local pressure at the center of the contact will most likely induce "squeeze-out" of the lubricant in all environments. The kinetics of this

process, however, are likely to vary. The dynamics of the hydrophobic monolayer in the aqueous environment of the measurements may indeed be slower due to the necessary changes in hydration of the squeezed out film, resulting in slower "squeeze-out" and therefore better surface protection by the film during shorter contact times. This is consistent with the notion that the increased adhesion force arises from substrate interactions, though in a sharply rate dependent manner on the millisecond timescale. Lateral motion of the tip is also a possible contributing factor, the parameters of these experiments suggest a longitudinal tip motion on the order of 0.06 to 2 Å/s, alternatively 1 to 30 seconds per Si-O bond length. Sliding of the tip during the adhesion experiment can alter the chemical interactions between the tip and the surface at the instant of pull-off. Moreover, this longitudinal tip motion in conjunction with the surface roughness would increase the instability of the adhesive regime contact on the nanoparticle roughened surface, which could explain the dramatically lower pull-off force observed only for the roughened surface contact, as contact area arguments alone are insufficient to explain this result.

The surface chemical interaction occurring at SAM coated asperity contacts is critical to their tribochemical response. Bond formation at the silica-silica interface is exactly the type of interaction the SAM nominally prevents, and it leads to both degradation of the film and the underlying surface. This is a likely basis for the beginnings of wear and surface failure in OTS functionalized MEMS, where asperity contacts in the rough surfaces of these devices consists at least in part of direct silica interaction that leads to tribochemical degradation of the film and the interface.

5.3.2 Friction Response of SAM Coated Interfaces

A variety of friction responses have been observed for SAM coated contacts, and these responses can provide insight into the mechanical and chemical components of friction. A rather straightforward example of the varying friction response of SAM coated interfaces was demonstrated by Carpick *et al.*[281] They examined the friction response of OTS SAMs in various configurations by AFM. These configurations include the SAM applied only to the AFM tip, only to the surface examined, to both, and to neither, and

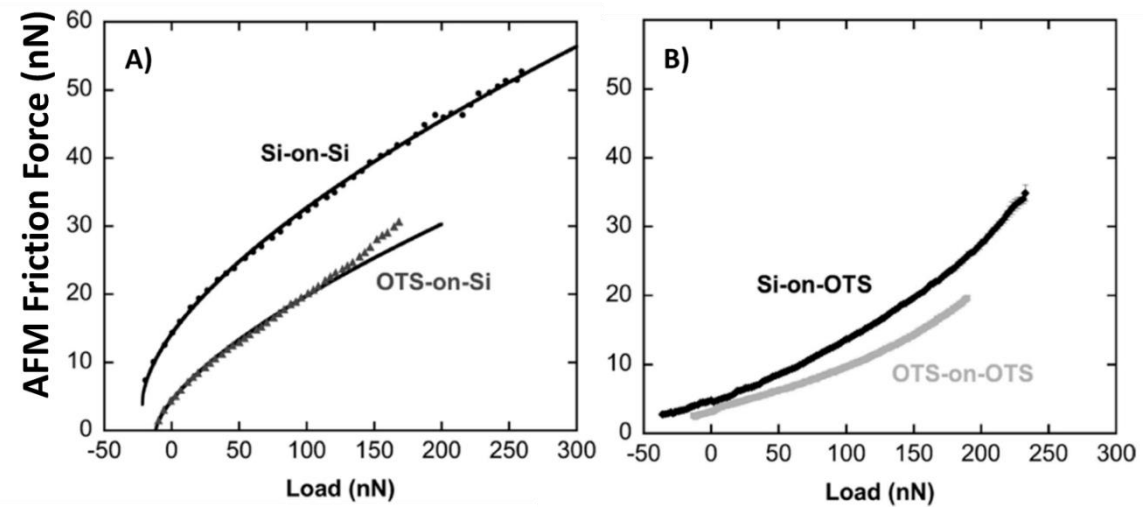


Figure 5.4. The AFM friction response of OTS SAMs depending on the configuration of the SAM in the contact. For contacts in which neither surface was functionalized, or only the tip was functionalized (A), responses consistent with single asperity friction laws were observed, indicated by the sublinear behaviour and the general COS fits shown. For contacts in which only the surface, or both the surface and the tip were functionalized (B), linear behaviour at low loads to superlinear behaviour at greater loads was observed, behaviour not predicted for contacts with a constant interfacial shear strength and friction coefficient. Adapted with permission from Flater, E. E.; Ashurst, W. R.; Carpick, R. W. "Nanotribology of Octadecyltrichlorosilane Monolayers and Silicon: Self-Mated versus Unmated Interfaces and Local Packing Density Effects". Langmuir. 2007, 23, 9242-9252.[281] Copyright 2007 American Chemical Society.

their results are summarized in Figure 5.4. For bare contacts and for contacts in which only the AFM tip was coated with the OTS SAM, the friction response was observed to be consistent with the laws of single asperity friction. In cases in which the surface was functionalized, the behaviour was not observed to correspond with any known friction laws, though at sufficiently low loads the response was relatively linear, consistent with Amonton's law.

Consideration of these friction responses with the pressure profiles depicted in Figure 5.3, it is clear why these different behaviours are observed. When the surface is not functionalized, substantial direct contact between the silica interfaces occurs. This drives both tribochemical pathways at the silica interface and strain mediated dissipation that gives rise to the contact area dependence observed and shown in Figure 5.4A. When only the AFM tip is functionalized, a similar response is observed, though it is likely at low loads the tribochemical pathways are inhibited by the film resulting in a uniformly smaller slope. At ~150 nN however, the friction response begins to approach that of the bare asperities indicating that the film is likely being sheared off of the AFM tip or at least displaced from the contact.

Perhaps the more interesting behaviour, is that observed in the case when the flat opposing surface was functionalized. The linear friction response at low loads is consistent with the three term friction law dominated by the load dependent term, reasonable if the interfacial shear strength is substantially reduced by the SAM coating on the surface. The source of the superlinear behaviour is unclear, however. A similar

example of this sort of friction response was observed for polystyrene near the glass transition temperature and coated with a hard polymeric layer.[303] The authors attributed the superlinear friction response with dissipation pathways associated with the α-transition

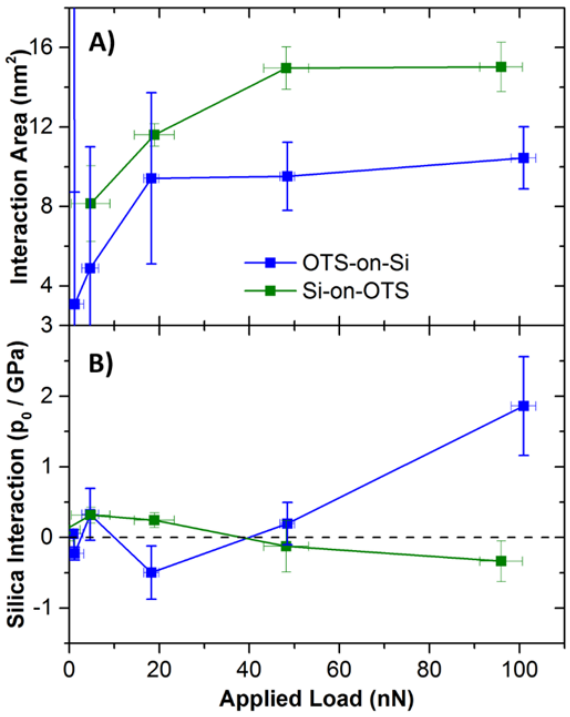


Figure 5.5. Total interaction area (A) and peak silica interaction pressure (B) for OTS-on-SiO and SiO-on-OTS contact configurations for asperity-flat interactions. For tip-functionalized contacts, the interaction area appears to continue rising at higher loads, and direct substrate interaction is indicated by increasing pressure at the silica-silica interface. For surface functionalized contacts, the total interaction area appears to saturate at about 50 nN, and the silica-silica interface does not appear to provide any appreciable contribution to the contact pressure. Errors in these measurements result from the triplicate sampling of the contacts in different orientations and therefore represent inhomogeneity of the surfaces and the films. More diffuse pressure distributions at lower applied loads generally results in much greater uncertainty particularly at low applied loads.

of the polymer. They hypothesized that the opening of this pathway coincided with sufficient strain propagation through the hard overlayer to induce the local phase transition. The question that must be asked, then, is what is the newly opened dissipationpathway that gives rise to the superlinear behaviour? And does the substrate play a direct role or is it a pathway localized within the film such as the purported "molecular plowing" mechanism?

The most direct comparison of these contact configurations are those in which only one surface was functionalized. The contact area as a function of applied load, for the tip functionalized and surface functionalized simulated contacts, are shown in Figure 5.5A. When the surface was functionalized, the total area of interaction saturated even at relatively low applied loads, not surprising given the softness of the film. When only the tip was functionalized, a gradual rise in the contact area was still observed as the load approached 100 nN. The fact that the total area of interaction saturated so quickly for the surface functionalized contact clearly indicates that the behaviour in this case is not consistent with single asperity friction laws, but must be dominated by the load dependent response. A more stark contrast is observed between these two configurations if the silica interaction is considered. Unfortunately the pressure profile is too diffuse to fit the contact area precisely, but the peak pressure at the silica-silica interface is shown in Figure 5.5B. For the tip functionalized contact, a clear rise in the interaction pressure between the substrates occurs at about 40 nN, while no repulsive contact occurs at the silica-silica interface up to 100 nN for the surface functionalized contact.

From these results, it would appear that substrate interactions do not contribute directly

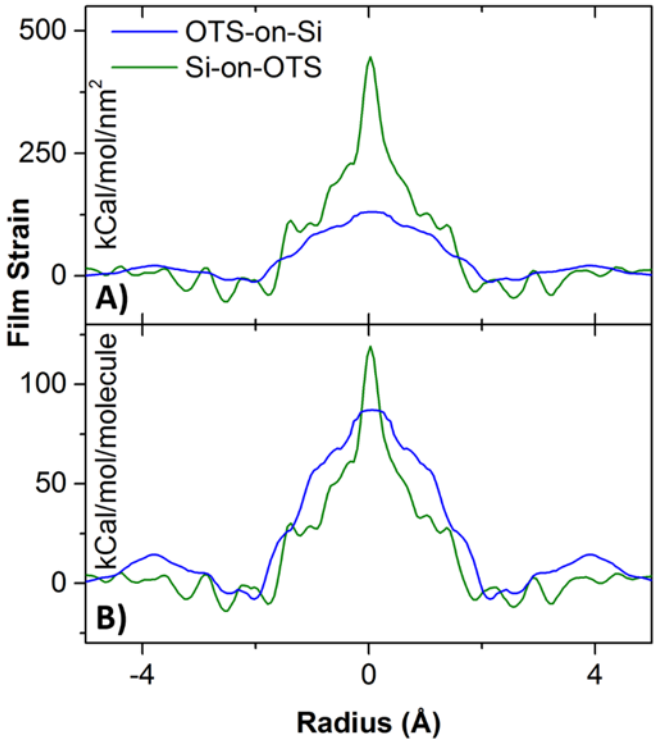


Figure 5.6. Radial profiles of the film strain energy per unit area (A) and per molecule (B) for surface-functionalized and tip functionalized contacts at 100 nN applied load. Corresponding pressure profiles were shown in Figures 2A and 2C respectively. An OTS film bound to the surface is observed to absorb much greater strain then a film adsorbed to the asperity surface. Interestingly, the magnitude of strain appears to correlate with the film packing density as the per molecule strains are similar. Moreover, the strain felt by the film near the center of the contact is near that of the Si-O bond strength, indicating low barriers to bond scission and tribochemical wear of the OTS film under these conditions.

to the frictional response of flat surfaces with a densely packed SAM, even at fairly high pressures of a few GPa, though closer examination of the strain in the film can provide some insight into the processes that likely play a role. Figure 5.6A depicts the strain energy in the film for both contact configurations. The strain energy density is far greater for the surface functionalized contact. During sliding on a SAM functionalized surface,

this strain energy will evolve on the leading edge of the tip, and fall on the trailing edge of the tip, and the energy dissipated is related to the magnitude of this strain energy. When only the tip is functionalized, lower strain magnitude is observed, though this is not surprising as the molecular packing density is lower on the tip. Furthermore, in the SAM-on-tip configuration, compression and decompression of the SAM is not relevant because the SAM slides with the asperity, so such a dissipation pathway is unlikely. Interestingly, the strain energy normalized to the packing density of the SAMs on these surfaces, shown in Figure 5.6B, is nearly identical for both contact configurations. The magnitude of the strain localized at the center of the contact is sufficient to promote bond cleavage within the film and at the film-silica interface. While these strains are nominally present in the entirety of the film within the contact, it is likely to localize primarily in the least rigid interactions, including the Si-O and Si-C bonds binding the film to the surface and holding the film together, as well as much weaker hydrogen bonding interactions that may be present in the film. This dramatically reduces the activation barrier to bond scission, which in conjunction with environmental factors like surface moisture[304] would induce the onset of tribochemical wear of the SAM. That this wear is not apparent here is a limitation of the force fields employed, which cannot accurately model bond scission but can be used to identify where bond scission is likely to occur.

In a top-down investigation of SAMs in a MEMS device contact with an apparent pressure of ~10 MPa, it was observed that there is easily sufficient compressive strain energy to promote removal of the molecules from the surface,[175] in fact showing that the strain energy density is nearly 10 times the bonding energy density of the molecules to the

surface. Applying the Greenwood-Williamson model, the mean contact pressure was estimated to be 13-16 GPa, in line with the OTS functionalized asperity-asperity contacts examined here, however we found that the actual strain distribution in the film is on the same order of the bond energy density. The disparity lies in the fact that the strain is partitioned between the film and the silica, but the conclusion is ultimately the same. Simulation demonstrates that the reaction barrier to bond scission is significantly diminished for the *average* asperity contact within the macroscale contact, which implies

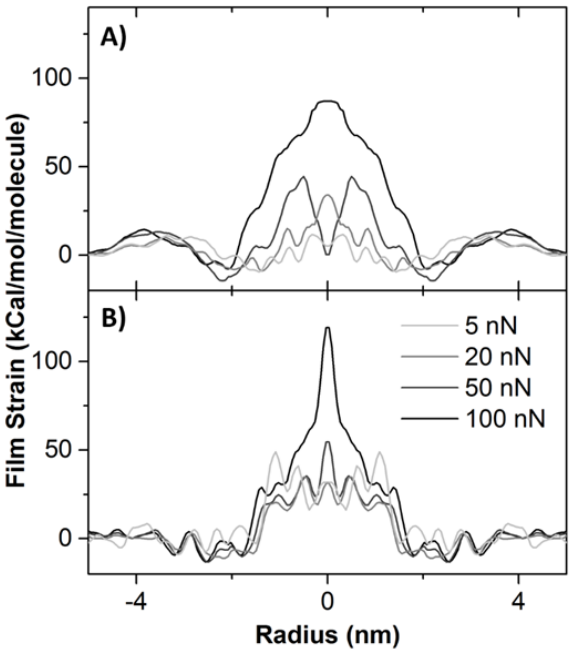


Figure 5.7. The film strain energy per molecule for the tip-functionalized (A) and surface functionalized (B) contact simulations as a function of applied load. While the tip-functionalized configuration shows increasing film strain with increasing load, the surface functionalized configuration shows relatively static film strain up to 50 nN, then a dramatic increase in strain localized to the center of the contact.

that significant film removal would occur at the contacting interface.

Interestingly, depending on to which surface the film is attached, the onset of strain in the film behaves uniquely. Figure 5.7 shows the evolution of film strain with load for the OTS SAM attached only to the tip or only to the surface. For the SAM-on-tip configuration, the film strain increases relatively gradually with increasing load. For a high density SAM on a flat surface, however, the film strain remained fairly static at loads below 50 nN, and changed dramatically at 100 nN. This sharp transition in the manner in which the SAM bears the loads of the asperity is likely related to the shift in friction response at greater loads, suggesting different dissipation processes within the SAM, and given the magnitude of these strains these dissipation mechanisms likely include chemical degradation of the SAM.

In these cases, it is clear that substrate interactions are relevant when the film is sparse, as is typically the case of SAMs on rough surfaces. Because the film on the asperity is so sparse, the evolution of strain in the SAM is likely minimal compared to the bulk strain experienced in these contacts, indicating that the mechanical coupling of the substrates through direct interaction is the determinative factor in the friction response. The SAM only acts to interfere with the interactions between the surfaces, lowering the mechanical coupling, and inhibiting tribochemical pathways, which reduces the shear strength and friction coefficient respectively. As these interactions lead to tribochemical wear of the film, however, the friction response reverts to that of the unfunctionalized interface. This is likely the dominant friction mechanism in applications of silane-derived SAMs in devices, as surface roughness is a primary factor in the extent of functionalization.

When the SAM is applied to a flat surface, at least prior to perturbation by the tip, the density of the molecules on the surface is sufficient to prevent intimate substrate interaction. At sufficiently high loads, a sharp transition it the magnitude of the strain is observed, resulting in new dissipative mechanisms available during sliding. In addition to the configurational changes that would give rise to this change in strain distribution, the sharp rise in strain energy also opens up chemical pathways of film removal from the surface. The former would likely correspond to a change in the interfacial shear strength, and the latter a change in the friction coefficient, and the simultaneous increase in both of these coefficients is likely the source of superlinear friction response with load for SAMs under moderate pressures. Unfortunately, a key observation here is that the dissipation and wear mechanisms for silane-derived SAMs in the laboratory and in applications on technologically relevant (*i.e.* not atomically smooth) surfaces are quite different, but the key differences are primarily isolated to silane-based monolayers, as other SAMs and boundary lubricants are relatively less sensitive to preparation conditions and surface morphology.

5.4 Conclusions

Substrate interactions were observed to play a particular role in the modification of surface forces for SAMs of low packing density. This is likely a driving factor in the failure of covalently bound SAMs like OTS to inhibit wear in MEMS, as surface passivation was likely not as extensive and robust on the naturally rough surfaces of these devices, compared to more ideally prepared SAMs on flat surfaces. For densely packed SAMs on flat surfaces, it was found that direct substrate interaction was relatively

negligible at pressures of a few GPa, though a marked shift in the compressive strain within the film was observed that would lower reaction barriers to bond scission and film wear. During sliding, the SAM must compress on the leading edge, and decompress on the trailing edge, and changes in this compressive strain would likely lead to variations in the shear strength and coefficient of friction of the sliding contact, leading to the superlinear friction response observed in single-asperity friction measurements of SAMs at high applied pressures, in addition to the likely increased role of dissipation pathways associated with film wear.

CHAPTER VI

FABRICATION OF NANOCONFINED MOLECULAR STRUCTURES AND THE IMPACT OF CONFINEMENT ON THE CHARGE TRANSPORT MECHANISMS OF PORPHYRIN ENSEMBLES

6.1 Introduction

In this chapter, the effects of molecular aggregation in fabricated molecular assemblies of a hydrocarbon tethered zinc porphyrin thiol was investigated. An initial investigation of the charge transport through these molecules singly and in small clusters in mixed monolayers was first performed, establishing in conjunction with other studies,[305] that the coordination of zinc into the porphyrin macrocycle promotes molecular aggregation and a resulting transition from tunneling to charge hopping transport. To take advantage of this property in a controlled manner, nanografting was employed to fabricate molecular ensembles with specified dimensions. Determination of the suitability and optimal conditions under which fabrication of these porphyrin nano-islands is discussed, in addition to the results of local confinement on the fabricated zinc porphyrin thiol.

6.1.1 Lateral Charge Delocalization as a Means to Transition from Tunneling Transport to Charge Hopping Transport in Molecular Junctions

The use of molecules to modulate the flow of current at interfaces is not a new concept,[42] but it has been fraught with many challenges. There are several advantages of molecules over solid state materials for the control of charge transport at interfaces, including improved thermal properties, [306] virtually limitless options via chemical

synthetic design, and the opportunity for "bottom-up" design and fabrication.[307] Among the many challenges molecular/organic electronic devices face however, perhaps the most significant is the specific connection between chemical structure and the resulting conductive characteristics of a molecule once it is assembled on a surface. This can be considered in terms of current density, or in specific, device-like characteristics such as current rectification,[42,43,308] conductance switching,[309-312] and non-differential resistance.[44,46] Rectification of carrier flow is of particular significance at a variety of interfaces, including solid state device interfaces,[313,314] photocatalytic surfaces,[315,316] and dye sensitized solar cells,[317] but even this relatively simple behavior has proven difficult to reliably and effectively achieve. The tunability of conductive properties has been observed for molecular wires like OPEs,[49,318] species which are highly conjugated and typically directly bound to the electrode, providing strong electronic coupling between molecular and electrode states. Due to this strong electronic coupling, however, features like persistent switchable conductance[312] must arise from structural[309,319] or environmental changes.[320] While other features like current rectification,[82] charge storage[83,84,321] and redox-based switching[310,311,322] are generally not feasible.

These limitations can be alleviated by decoupling the more conductive features of a molecule, such as aromatic rings or coordinated metal ions, from the electrode surface.[323] Such electronic decoupling can be achieved with solid state dielectric materials like oxides or salts,[127] or by directly incorporating a saturated hydrocarbon chain. [76,321] By decoupling the electronically active parts of the molecule from the electrode, persistent electronic changes like reduction and oxidation may be achieved,[43,85,322] vastly increasing the

potential functionality. Unfortunately, isolating these molecular functional groups from the electrode effectively decreases their capacitance,[84] such that the primary mode of transport is in many cases simply through-bond tunneling. This renders the junctions largely insensitive to any specific chemical modifications because the most determinative factor in the magnitude of a tunnel current is the junction length.[79-81] Variation in barrier height through chemical modification (e.g. altering molecular chain length) has limited effect, and the control of current flow through spatial variations of the barrier height via chemical functionalization is similarly seen to be ineffective, with rectification ratios between positive and negative bias for example, generally being limited to ~ 20 in the tunneling transport regime,[82,308,324] several orders of magnitude smaller than can be achieved with semiconductors.

In previous studies of transport in a hydrocarbon tethered porphyrin thiol on Au (111), a similar conclusion was reached.[325] In this molecule, the transport efficiency was dominated by the hydrocarbon tether, indicating that transport by off-resonant tunneling is dominant and that chemical variation of the otherwise highly modifiable porphyrin macrocycle would have limited impact on its transport properties. Porphyrins are of significant interest however for modulating transport properties at interfaces, due to their small HOMO/LUMO gap and accessible redox chemistry, their prevalence in light harvesting systems, and their ability to facilitate efficient charge transfer.[326,327]

By designing molecular systems that shift away from tunneling as the dominant mechanism of charge transport however, greater sensitivity to the chemical structure can be achieved. Transport via charge-hopping, for example, has been purported to increase

the functional relationship between molecular structure and transport characteristics, bringing the rich chemical changes that can be achieved synthetically back into play as a means to tune charge transport at interfaces and providing a robust means of enhancing CMOS technologies through advantageous chemical modifications. In situations where charge hopping can be emphasized, conduction depends greatly on the chemical potential of the molecular states within the junction,[77,84] and the relationship between chemical structure and chemical potential can be better understood, and can give rise to sharper response to applied bias or gate bias. Accomplishing this with defined molecular systems however requires one to overcome the challenge of stabilizing high charging energies on a molecule, such that a minimum molecular size is required.[84] Examples exist of large molecules facilitating charge transfer via sequential electron hopping,[76,77,310,311] but this approach substantially increases synthetic complexity, which can have a negative impact on chemical tunability. Such challenges can be ameliorated by a molecular design that facilitates nearest-neighbor interactions to drive molecular assembly through the controlled formation of aggregates, such that control of domain size proffers an additional lever by which charge transport behavior may be tuned. Here, lateral delocalization of charged states within a film is a more promising avenue by which the charge hopping based transport can be achieved,[64] and this leads to more efficient transport at the interface.[328]

6.1.2 Charge Transport in Zinc Porphyrin Thiol Molecules

Mixed monolayer studies of the conductivity of the zinc porphyrin thiol, depicted in Figure 6.1 were conducted. In these studies, molecular conductivity, in terms of the

tunneling efficiency, was determined from topographic images of the molecules embedded in an alkanethiol matrix. The double-layer tunnel junction was used to determine the molecular conductivity, which requires a detailed analysis of the geometric structure of the molecules in the mixed monolayer, this was achieved through the use of DFT geometry optimization and electronic structure calculations, as well as AFM imaging to determine the structure and orientation of the molecules in the film. With this information, STM imaging of the mixed monolayer was used to determine the conductive characteristics of single molecules and small, randomly formed aggregates.

Electronic Structure Calculations

DFT[329] was used to calculate the molecular and electronic structure of the zinc porphyrin thiol. Geometry optimization and single point energy calculations of the zinc porphyrin thiol were performed with the Gaussian 03 computational suite.[330] The TPSS DFT functional[331] was used for geometry optimization, single point energy calculations, and vibrational spectra. Calculations were performed using the 6-31+G(d') basis set[332] plus the D95 full double zeta basis set[333] for the zinc cation, and orbital population analysis was performed using the AOMix software package.[334]

The calculated gas phase optimized structure of the zinc porphyrin thiol molecule, and a model of its insertion geometry into the alkanethiol matrix, are shown in Figure 6.1A and 6.1B, respectively. The optimized structure of the zinc porphyrin thiol is similar to that of the free base analog studied previously,[325] in which the tetrafluorophenyl ring is canted nearly perpendicular at ~84 degrees with respect to the macrocycle, while the less sterically hindered pyridyl substituents are canted only 65 degrees.

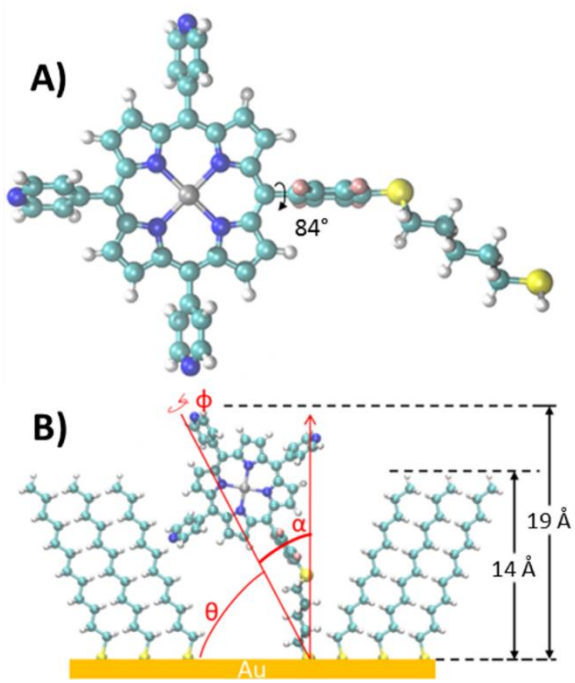


Figure 6.1. (A)The optimized structure of the thiol-tethered zinc porphyrin molecule as determined from DFT calculations, where sky blue atoms are carbon, white are hydrogen, blue are nitrogen, pink are fluorine, yellow are sulfur, and gray is the zinc(II) ion. In (B), a model of the insertion geometry of the porphyrins in the alkanethiol is depicted. The rotational angle φ is such that the porphyrin would lie flat on the surface if tilted all the way over, and α is determined from AFM microscopy to be ~ 30-45°, corresponding to a height difference of ~ 3-5Å relative to the SAM matrix.

The HOMO-LUMO gap for the zinc porphyrin thiol was calculated to be 1.99 eV which is slightly larger than 1.89 eV calculated for the free base analog. The HOMO energy level of -5.37 eV is close to the reported Fermi level of the Au(111) surface, suggesting a small charge injection barrier, though the absolute energy scale of DFT

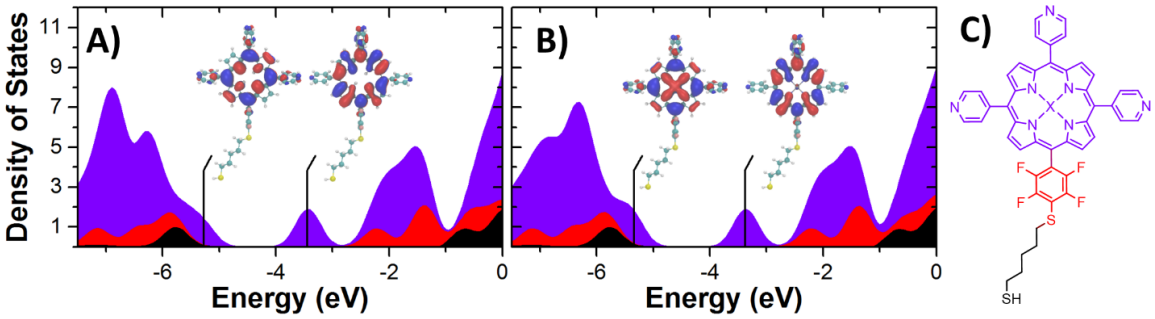


Figure 6.2. The partial density of states for the freebase (A) and zinc substituted (B) porphyrin are depicted. The colors correspond to the coloration of the chemical components shown (C), and the HOMO and LUMO orbitals of each molecule are also depicted. The frontier density of states are nearly identical, with the majority of the frontier state density existing only on the porphyrin macrocycle.

calculations can be unreliable. As shown in Figure 6.2, for both the zinc metalated and freebase analog of this molecule, there is no frontier state density present on the fluorophenyl linker group, a result of the aforementioned, near-perpendicular dihedral angle of this ring with respect to the macrocycle. The lack of extension of the pi system beyond the porphyrin macrocycle, and the existence of the hydrocarbon tether, effectively decouples the porphyrin macrocycles from the metal surface when adsorbed in the standing geometry, providing an ideal configuration of a double-barrier tunnel junction.

Analysis of Molecular Orientation by AFM

AFM images were acquired with an Agilent 5500 AFM. All AFM images were acquired in contact mode under ethanol using commercially available Si_3N_4 AFM tips (Bruker AFM Probes, MSCT, Sunnyvale, CA) with nominal tip radii of ~10 nm and nominal spring constants ranging from 0.03 - 0.1 N/m and applied loads of ~0.1 nN to minimize compression of the porphyrin molecules during imaging.

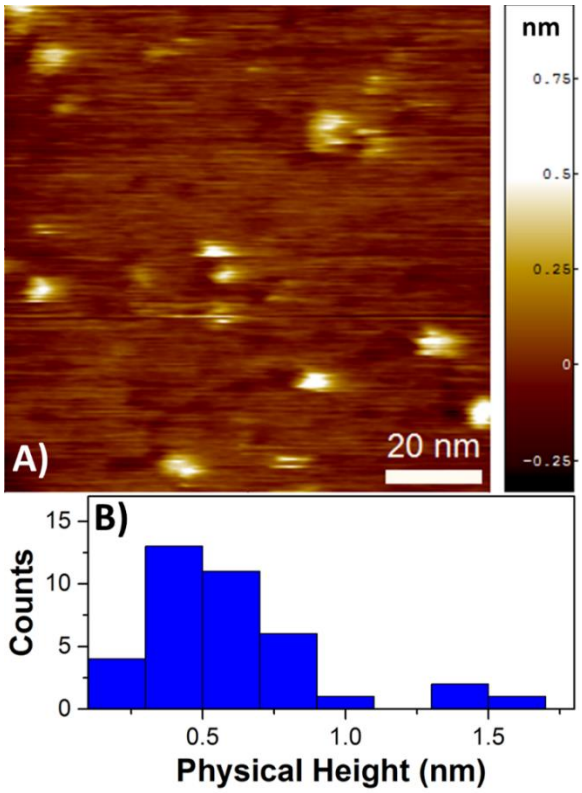


Figure 6.3. An AFM topographic image (A) and statistical distribution of physical heights (B) measured by AFM for the zinc porphyrin thiol clusters. The porphyrin protrusions appear as bright spots in the topograph, and statistical analysis of observed heights produce an average height of 5±2 Å above the DDT matrix. The sample shown was soaked in a 0.1 mM zinc porphyrin thiol solution for 3 days, and was imaged with an applied load of 25 pN in ethanol.

Because observations of the electronic properties of these molecules by STM are intrinsically coupled with the structural configuration of the molecules on the surface, and to confirm that the molecules are in a standing geometry, it is necessary to determine the structure of the porphyrin molecules in the mixed monolayer. AFM was used to determine the physical height of the porphyrins protruding from the alkanethiol SAM (Figure 6.3A), with the observed distribution of heights shown in Figure 6.3B, yielding an average

physical height of 5±2 Å above the DDT matrix. This implies a molecular tilt, α, of 30° - 45° from the surface normal, which is up to 15° more canted than the typical alkanethiol film, and is likely due to the steric interactions of the porphyrin and that the molecules insert predominantly at defect sites within the film.

Conductance of Single Molecules and Small Clusters

STM was performed to characterize the conductance of single molecules and small molecular clusters in the alkanethiol matrix. An Omicron UHV-VT STM system with a typical base pressure < 3 x 10^{-10} Torr was employed and mechanically cut Pt/Ir (80/20) tips were used for imaging. Images and STS were collected with a setpoint current of 20 pA and tip bias voltage of 1.4 V.

Prior mixed monolayer studies of the freebase porphyrin thiol molecule[325] in the DDT matrix exhibited a preference for insertions of single molecules to clusters of only a few molecules (typically 3 – 4) as determined by the width distribution. To compare the transport properties of the single metalloporphyrin or clusters of only a few metalloporphryins, the DDT matrix was immersed in dilute solutions of the zinc poryphyrin thiol in DCM. A representative STM topographic image of the imbedded metalloporphyrins is shown in Figure 6.4A, with the height and width statistics from several images in Figures 6.4B and 6.4C, respectively. The widths of the clusters are consistent with insertions of one to a few molecules, and the primary population of the apparent heights of 0.5±0.2 nm is also comparable to the apparent height measured for the freebase analog.[325] The secondary population with apparent height of 1.5 nm indicates more conductive porphyrin clusters, and this population can be amplified by increasing

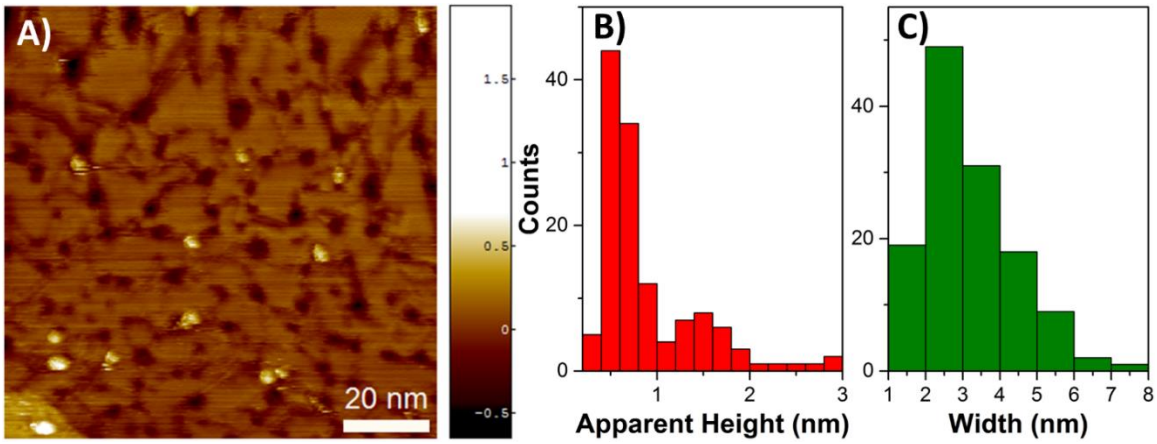


Figure 6.4. (A) STM topographic image showing porphyrin thiols inserted into the DDT matrix along with distributions of the apparent height (B) and width (C) of the embedded zinc porphyrin thiols collected over a number of images. The sample shown was immersed in a 0.1 mM zinc porphyrin thiol solution for 3 days. (bias = 1.4 V, I = 20 pA).

the concentration of the porphyrin solution. The double-layer tunnel junction model[335] was used to determine the overall tunneling efficiency of the molecules by the equation:

$$\beta_{ZnPn} = \left[\beta_{DDT} h_{DDT} - \alpha\left(\delta h_{STM} - \delta h\right)\right] h_{ZnPn}^{-1} \quad \text{..... (6.1)}$$

Where β_{DDT} is the tunneling efficiency of the DDT matrix (1.2 $Å^{-1}$)[335], h_{DDT} is the thickness of the DDT film (14 Å), α is the tunneling efficiency of vacuum (2.3 $Å^{-1}$), δh_{STM} and δh are the protrusion heights of the zinc porphyrin thiol measured by STM and AFM respectively, and h_{ZnPn} is the height of the molecule determined by summing h_{DDT} and δh. The tunneling efficiency for the single metalloporphyrin was found to be 0.9±0.1 $Å^{-1}$. Assuming the hydrocarbon tether has the same tunneling efficiency as the alkanethiol matrix, and adjusting the tunneling efficiency equation to determine only the macrocycle (MC) tunneling efficiency:

$$\beta_{MC} = \left[\beta_{DDT} \left(h_{DDT} - h_{tether} \right) - \alpha \left(\delta h_{STM} - \delta h \right) \right] h_{MC}^{-1} \quad \text{..... (6.2)}$$

Where h_{MC} is 17 Å projected along the surface normal, corresponding to the length from the 10-(4-pyridyl) nitrogen to the sulfur atom linking the 20-(4-thiophenyl) ring to the hydrocarbon tether. This yields a tunneling efficiency of 0.7±0.2 $Å^{-1}$ for the macrocycle, similar to that observed for other aromatic, highly conjugated molecules[78] but much greater than that observed for zinc(II) porphyrin nanowires.[336] Though the total tunneling efficiency of the molecule does differ from the freebase, the apparent height measured by STM was nearly identical for the two molecules. The primary difference is the measured physical height, for which there is a significant amount of uncertainty, and in fact the 7 Å physical height measured for the freebase is within the ±3 Å distribution of physical heights measured for the zinc complex. The error in the physical height arises from the dependence of the measurement on how the AFM tip interacts with the porphyrin thiol and the SAM, which can vary with tip shape and applied load. Though similar tips were used to measure the physical heights of the freebase and zinc analogues, there are variations in tip sharpness, and the forces used during imaging were the smallest achievable for the instruments (10 pN - 100 pN), thus precise control over the applied load is difficult to achieve. It is unlikely, however, that the addition of zinc to the macrocycle has a dramatic impact on the insertion geometry, such that comparison of the apparent heights is a reasonable and direct means of comparing the tunneling efficiency of these molecules, and for which no difference is observed, indicating that addition of the zinc ion has no impact on the conductivity of single molecules inserted in the SAM matrix..

Tunneling efficiency measurements provide insight into the conductivity of the

molecules at a given bias, 1.4 V in this case. STS measurements can be used to quickly examine differences in conductivity over a range of applied bias. I(V) spectra of the single and few molecule clusters of the metalloporphyrin are shown in Figure 6.5 imposed over the spectra of the DDT background. At low bias, the I(V) spectra were found to be similar to the alkanethiol, with greater current magnitudes observed at greater bias suggesting resonant tunneling contributions from the metalloporphyrin. The range over which the curves are similar is ~2.5 V, a value similar to the molecular HOMO-LUMO gap. This suggests that the Fermi level lies directly between the HOMO and LUMO states, however DFT calculations, conductance measurements, and conductance state switching of large molecular aggregates at positive tip bias suggest that the molecular HOMO lies close to the Fermi level. An alternative explanation is that the greater electric field, in conjunction with the greater polarizability of the porphyrin macrocycle compared to the

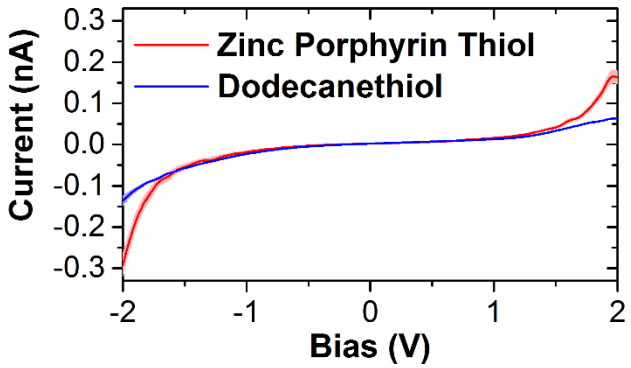


Figure 6.5. Representative I(V) spectra of single and small clusters of the zinc porphyrin thiols compared to that of the DDT matrix, as averages of 288 and 300 individual spectra respectively. The light shading around the curves indicates the standard error range of the measurements.

alkanethiol background, gives rise to greater conductance at high bias. The similarities in spectrum shape and magnitude obtained for the metalloporphyrin and the freebase indicates that addition of the zinc ion has little or no impact on the charge transport characteristics. This result is consistent with a small change in the attenuation of the charge carrier wave function through the molecule resulting in minimal variation in charge transport as suggested by Whitesides and coworkers.[79 80]

The single molecule transport characteristics are consistent with a tunneling based mechanism, which is largely insensitive to chemical structure. This is supported by the fact that there is virtually no different in charge transport characteristics with coordination of the zinc cation, with both the freebase[325] and zinc coordinated analog exhibiting an apparent height of 5 Å. However, the secondary population of apparent heights observed in Figure 6.4B is unique to the zinc porphyrin thiol, and is attributed to enhanced aggregation due to the addition of the zinc cation. Lateral charge delocalization in these porphyrin islands facilitates a transition from purely tunneling based transport to transport via the more efficient, and likely more controllable charge hopping mechanism, and this transition in mechanism is supported by the observation of Coulomb blockade and persistent bias induced switching also observed for these molecules,[305] both of which indicate the ability of the molecules to store charge. The remaining focus of this chapter is the fabrication of zinc porphyrin thiol assemblies on surfaces with specified geometries. By controlling the domain size of these molecular islands, control of the lateral delocalization of charge states may be achieved, which in turn controls the energy spacing and availability of these charged states as pathways for molecular conduction. It should

therefore be possible to tune the conductivity of these molecular aggregates by controllably patterning them on the surface. In order to achieve this, first, the suitability and appropriate conditions for the AFM nanografting approach were determined, followed by generation of porphyrin islands of varying size, which were found to exhibit varied conductance.

6.2 Optimization of the AFM Nanografting Process for Fabrication of Molecular Islands

A variety of factors affect the quality of nanografted structures.[337] Quality implies that the grafted structure is well-ordered and uniform, with complete removal of the SAM matrix and replacement by the target molecules. Parameters that influence the quality of nanografted structures include the speed at which the tip is dragged across the surface in order to shear the matrix SAM from the surface, the applied force used to displace the molecules from the surface, the line spacing used to clear solid, 2-dimensional patches of the matrix film, and the solvent environment. The sliding speed during structure fabrication is nominally diffusion limited, however over the range of typical imaging speeds (0.1 – 2 μm/s),[136] this is not a factor, and practical factors like drift and piezo creep, which effect the final shape and line spacing of the nanografted structure, have a greater impact. Throughout the reported results, a sliding speed of 0.1 μm/s was used. The solvent environment primarily influences the rate of exchange or depletion of the SAM matrix which can result in random insertion of the target molecules into the matrix SAM.[338] The solubility of the SAM matrix and the target molecule in the solvent environment can also influence the necessary forces required to shear the matrix SAM off

the surface, as will be discussed later. Compatibility of the solvent with the instrument, the target molecule to be nanografted, and the SAM matrix must also be considered, for example water is excellent at minimizing SAM depletion, but water cannot dissolve most organic molecules that might be interesting or useful grafting targets.

The applied force and line spacing both depend on the geometry of the AFM tip used for fabrication. Sharper tips generally require lower applied loads, but tighter line spacing. This is because the shear history of the surface, which is directly related to the pressure and shear history of the surface, ultimately dictates the rate and completeness of removal of the matrix molecules from the surface. A peak applied pressure of ~10 – 15 GPa is necessary to clear the SAM from the surface, but because the exact tip structure is typically not known beforehand, the necessary force to remove the film is not known until attempts are made to do so. The sharpest AFM tips (R<5 nm) can graft at applied loads of 5-20 nN, and more blunt tips (R>10) can require 100 nN. Sharp tips are generally unstable however, and become blunter after extended use, with stable nanografting loads of 40-60 nN being typical.

In order to generate consistent and reproducible porphyrin nanostructures, it was necessary to optimize these parameters on the instrument employed, with the surfaces and SAM matrices and the tips employed. Line-spacing effects were examined by nanografting structures of 16-MHA. This molecule forms bilayer structures,[339] the mechanical integrity of which depends on the quality of the base nanografted structure. AFM is sensitive to mechanical properties of the surface, so that graft quality can be inferred from the height of these structures. Finally, with the development of methods for

efficient sample transfer and relocation, STM was used to examine the quality and completeness of nanografted structures of ODT, in order to confirm that complete matrix removal was achieved as this is critical for the formation of porphyrin ensembles.

6.2.1 Line-Spacing Effects in Nanografted Structures

To fabricate solid, 2-dimensional structures with the nanografting process, just as in imaging, the AFM tip must be rastered across the surface. The separation between raster lines traced by the AFM tip is a critical parameter in achieving complete clearance of the matrix SAM. Optimal line separation is dictated by the geometry of the tip and its mechanical properties, which dictate the contact deformation and distribution of pressure in the contact and the necessary applied load to remove the matrix SAM. The goal is to apply pressure as uniformly as possible throughout the nanografted region, with sufficient magnitude to remove the film. Many of these factors may be generalized by considering the line spacing in terms of the CRLS ratio, described by the equation:[340]

$$\mathrm{CRLS} = \frac{a}{LS} \quad \text{..... (6.3)}$$

Where LS indicates the line spacing, and a represents the contact radius. The contact radius can be determined assuming Hertzian mechanics as:[12]

$$a = \left(\frac{3RF_z}{4E^*} \right)^{1/3} \quad \text{..... (6.4)}$$

Where R is the radius of curvature of the tip, F_z is the applied load, and E^* is the reduced elastic modulus of the contact, determined by the equation:

$$E^* = \left(\frac{1-\nu_1}{E_1} + \frac{1-\nu_2}{E_2} \right)^{-1} \quad \text{..... (6.5)}$$

Where ν is the Poisson ratio, and E is the Young's modulus corresponding to the tip and surface materials. For a Au surface (ν=0.36; E=62.5 GPa) and a silicon nitride tip (ν=0.24; E=220 GPa), the reduced elastic modulus of the contact is 75 GPa. It is assumed that at sufficient load, the film is displaced and does not contribute to the contact mechanics, so its contribution to the reduced elastic modulus is ignored. Figure 6.6A depicts simulated

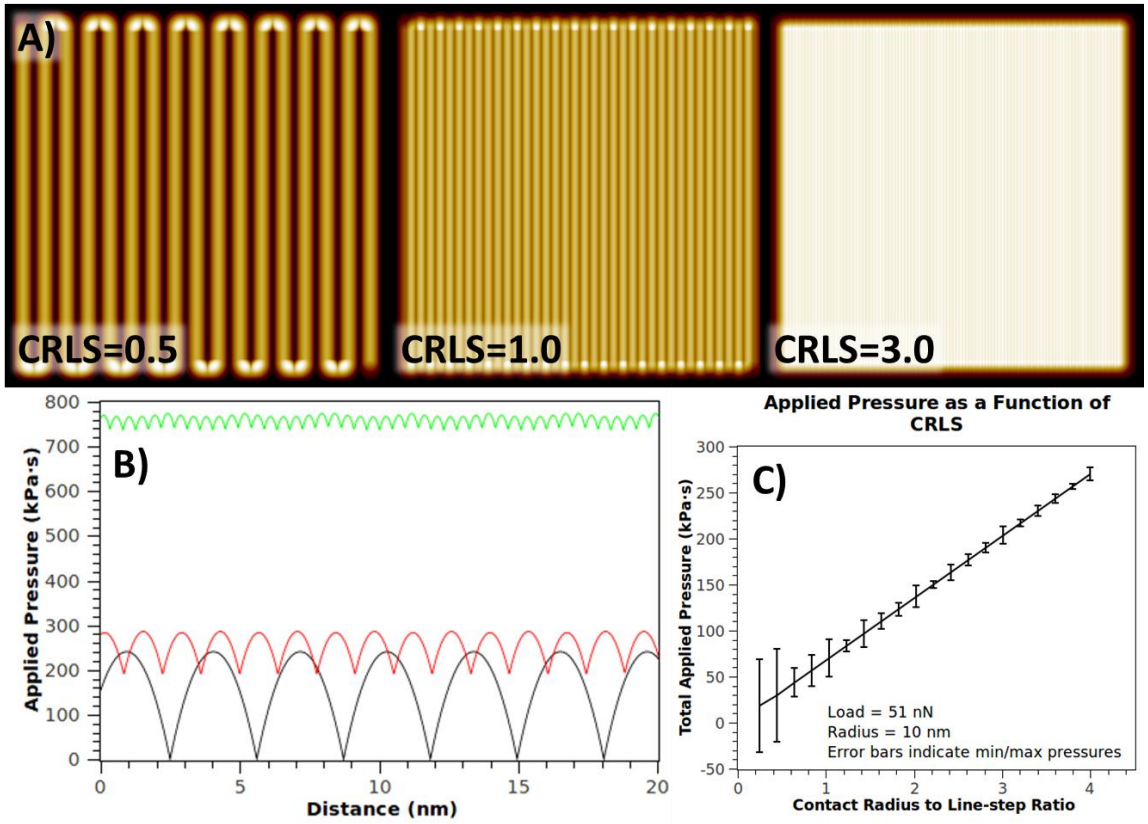


Figure 6.6. Simulated pressure histories for a 10 nm radius tip and an applied load of 51 nN for various CRLS ratios (A), with corresponding line traces (B) indicating the uniformity and magnitude of the pressure history on the surface. A linear trend between CRLS and pressure magnitude is observed (C) with decreasing variation, though variation does not considerably decline after a CRLS of 2.0.

pressure history maps for various CRLS values. A CRLS ratio of 0.5 corresponds to perfect line-spacing/contact area agreement, which means that any given point on the surface experiences contact with the tip once. Because the pressure distribution in the contact is not uniform, however, there is substantial variation in the pressure history over the surface, with the matrix likely removed from the center of the tip contact but not from the edges. Decreasing variation and increased pressure history is observed for increasing CRLS values, as viewed in the maps and the pressure history profiles depicted in Figure 6.6B. Figure 6.6C demonstrates the pressure magnitude dependence and variation for various CRLS values assuming an applied load of 50 nN and a tip radius of 10 nm. The magnitude of the pressure history is proportional to the CRLS ratio used, and the variation in pressure history decreases substantially up to a CRLS value of 2.0. Above this, gains in terms of decreasing variation are minimal with increasing line density, suggesting that 2.0 is an optimal value of the CRLS ratio.

Confirming experimentally that sufficient line density is used to completely clear the grafted area presents a challenge because AFM lacks the necessary resolution to clearly indicate if all molecules are removed from the grafted area. Defects in a positive height structure (one which protrudes from the matrix SAM) will not be visible due to a lack of resolution. Negative height structures, or using nanoshaving to confirm clearance of the SAM, are better, but subject to the ability of the tip to probe the entire depth of the well, and sparse layers left behind in the shaving process may lack the mechanical integrity to yield a response. To overcome these limitations, ensembles of 16-MHA were grafted into a DDT matrix. When the grafting is performed in a sufficiently high concentration($>10^{-8}$

M) of 16-MHA, bilayer structures have been shown to form, depicted in Figure 6.7.[339] Figure 6.8 depicts the height variations in nanografted 16-MHA bilayers as a function of the CRLS ratio used for features grafted in ethanol and 3P1P. In 3P1P, the full structure height of 1.3 nm was achieved at a CRLS ratio of less than 1.0. In ethanol, full structure height of 1.5 nm was achieved at a CRLS ratio of 3 or higher. The expected bilayer height, corresponding to a monolayer of 16-MHA tilted 30° to the surface normal and a second layer canted an additional 30°, is 1.6 nm.[341] The height measured by AFM reflects the mechanical stability of these bilayer structures, which provides a measure of the quality of the grafted structures because less complete removal of the DDT and replacement by 16-MHA will provide less stabilization of the bilayer structure via nearest neighbor interactions. The differences in the two solvent systems likely represent a

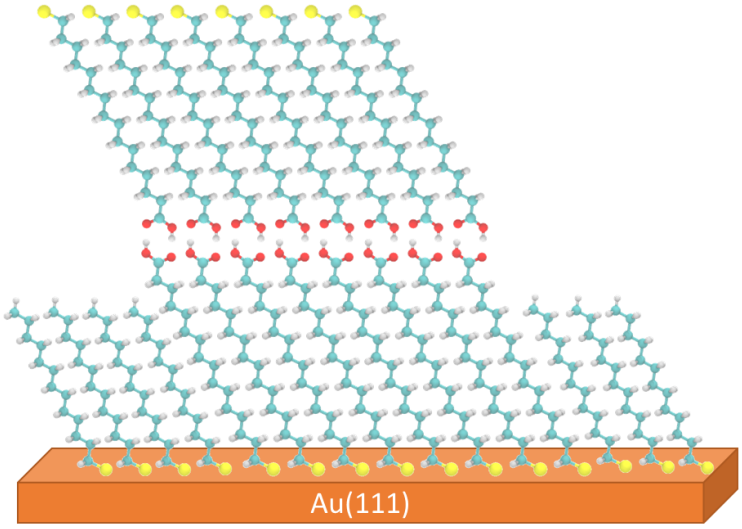


Figure 6.7. A model structure of the 16-MHA bilayer. Hydrogen bonding interactions of the carboxylic acid head groups stabilizes the formation of the bilayer.

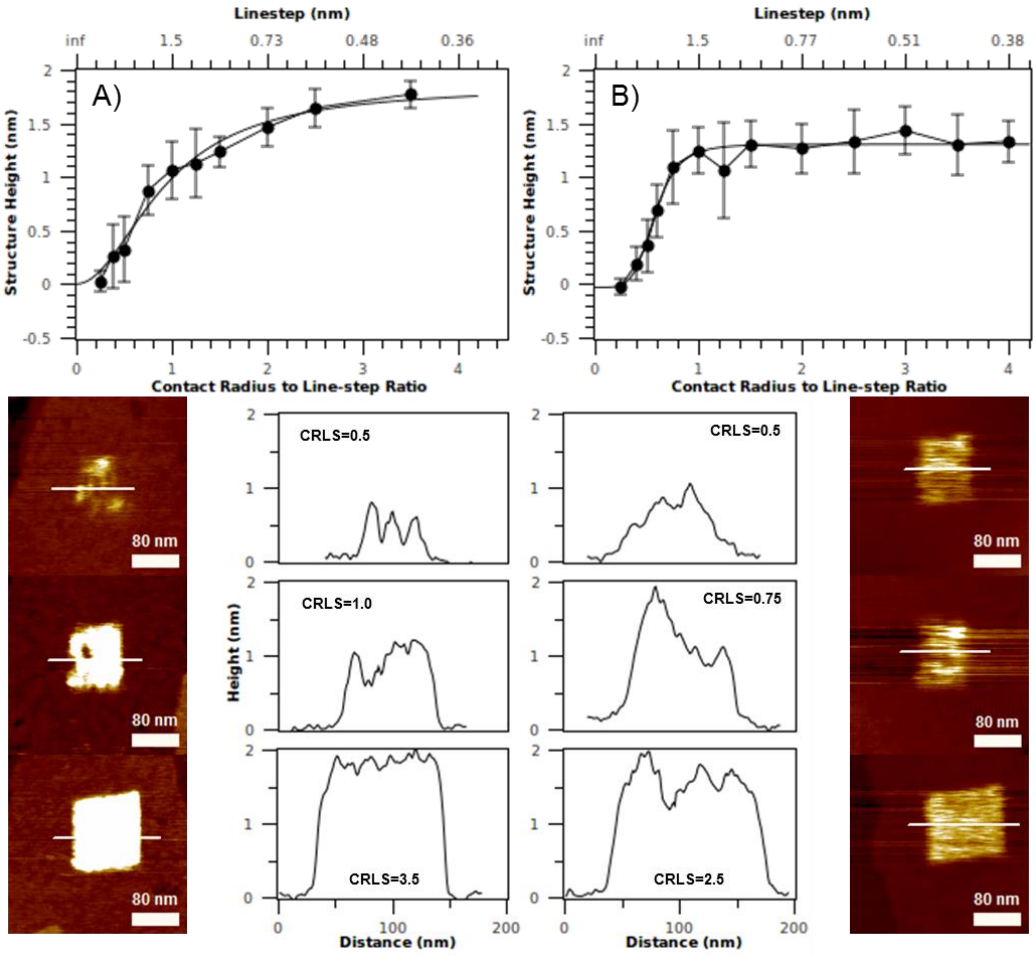


Figure 6.8. Dependence of structure height on the CRLS ratio used to graft 150x150 nm bilayer structures of 16-MHA in ethanol (A) and 3P1P(B). Corresponding structure topographies and lines traces at CRLS ratios before and after the transition from mono- to bilayer structures are shown.

combination of effects, including the ability of the solvent to facilitate the grafting process as well as its impact on the mechanical properties of the 16-MHA bilayer. While a lower CRLS ratio is required to achieve full structure height in 3P1P, there is much greater variation, and a lower overall height for the bilayered structure. This is likely due to the different solubilities of the long chain thiols in ethanol and 3P1P. Alkanethiols are

relatively insoluble in ethanol compared to more non-polar solvents,[338] so the DDT matrix will be more easily displaced from the surface and into the solution phase in a 3P1P environment, yielding a smaller minimum CRLS for complete displacement. Similarly, the greater solubility of the 16-MHA would result in greater solvent incorporation into the bilayer structures, possibly compromising its mechanical integrity and reducing its total height as measured by AFM.[342] In 3P1P, a CRLS ratio of 2.0 was more than sufficient to facilitate the formation of complete nanografted structures, while in ethanol marginal improvements can be realized by going to higher CRLS ratios, indicating that when grafting is done in non-polar solvents, like the DCM solvent used for fabrication of porphyrin islands, a CRLS ratio of 2.0 is sufficient to achieve complete structure formation.

6.2.2 Evaluating the Structural Quality of Nanografted Structures by STM

An area of concern regarding nanografted structures is their ultimate quality. It is unclear, for example, what impact the nanografting process has on the underlying Au(1111) surface, including whether the breakage and formation of new S-Au bonds induces reconfiguration of the surface atoms, or if the shear of the tip results in any damage to the surface that might be undetectable underneath the relatively soft nanografted features. Challenges of pattern relocation have generally inhibited researchers from being able to address this question directly, but it must be considered in the fabrication of nanografted porphyrin ensembles because their local environment and internal architecture will ultimately depend on their quality and completeness and the structure of the underlying Au(111) surface. With the development of techniques for transfer of AFM

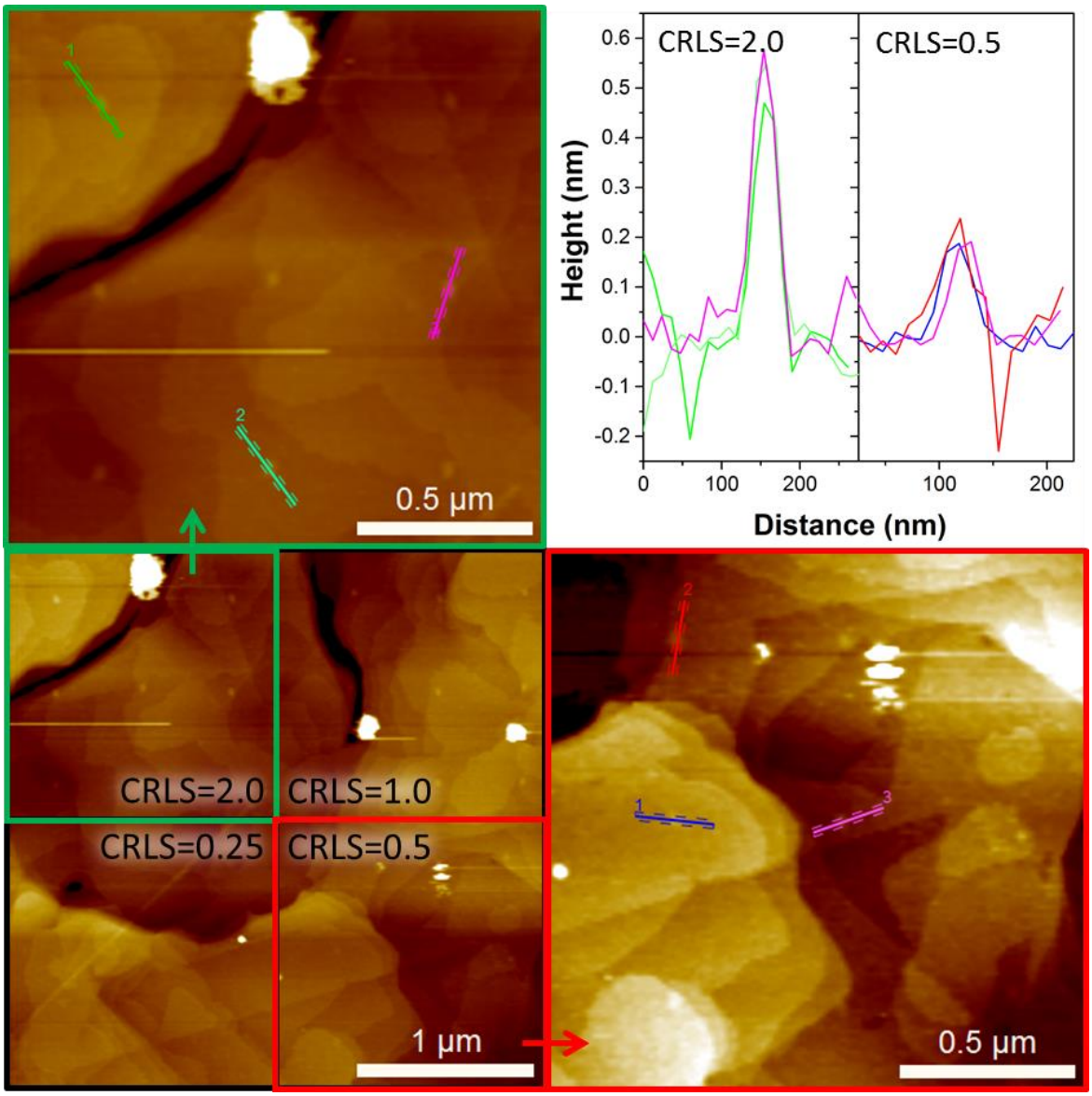


Figure 6.9. AFM topographic image of ODT nanografted structures in a DDT matrix (bottom left). Zoomed-in images of the CRLS=2.0 (top left) and CRLS=0.5 (bottom right) areas show distinct difference in the protrusion height, more clearly indicated with corresponding line traces shown in the top right.

nanografted structures and unambiguous relocation, it is possible to view, with molecular resolution, the quality of the nanografted structures as a function of the various grafting

parameters. For this investigation, ODT was used due to its similarity in structure to the DDT matrix and ability to self-assemble on its own, facilitating stable structures with little thermal fluctuations.

Patterns were grafted with CRLS values ranging from 0.25 to 2.0 using a silicon nitride tip with a nominal tip radius of 11±2 nm. The applied load used was 65 nN, corresponding to a peak contact pressure of 16 GPa. AFM topographies of the grafts are presented in Figure 6.9, with corresponding CRLS values indicated. The topographic images indicate that CRLS ratios of 0.5 and lower are insufficient to successfully graft complete structures, as there is little apparent modification of the surface in these regions, while clear protrusions of ~6 Å are visible for grafts with CRLS of 1 and 2, consistent with the prior investigation of 16-MHA bilayer nanografted structures and the expected protrusion height of the ODT from the background DDT.

An STM topography image of a nanografted feature with a CRLS ratio of 2.0 is shown in Figure 6.10. Interestingly, the structure appears as a 2 Å depression by STM. Though this result was not investigated in detail, this is expected to result from the substantial thickness of the ODT film, which dictates that the STM tip plow through the layer during imaging. This ploughing could affect the local structure of the SAM at the tip apex or the electronic coupling between the SAM and the tip, resulting in lower tunneling efficiency of the tunnel junction and an appearance throughout, suggesting complete clearance of the DDT matrix and replacement with ODT. Additionally, some reconfiguration in and around the grafted region was observed. The etch pits in the grafted well are larger than those in the surrounding matrix, suggesting that either removal of the original film, or

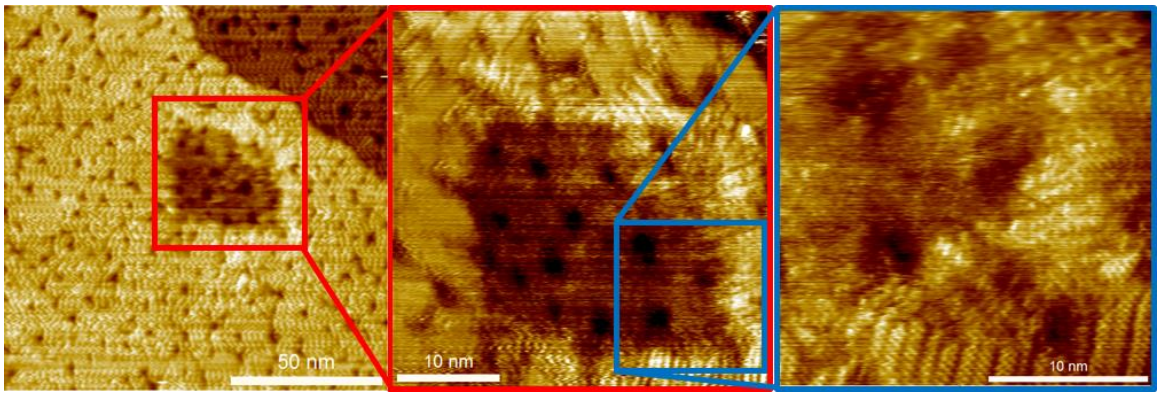


Figure 6.10. STM topography images of an ODT grafted structure in a DDT matrix. Variation in the etch pit structure and the phases of the surrounding DDT matrix are observed. The well is uniformly 2 Å in depth.

attachment of the incoming ODT molecules, results of a depression. The depth of the wells was observed to be reasonably consistent in changes to the Au(111) surface. Reconfiguration of the etch pits without the appearance of any extensive surface fouling is as good as might be expected. The S-Au bond has been found to be stronger than the underlying Au-Au bonds, so reconfiguration of the surface atoms is not a surprising result, and is likely unavoidable.

In addition to changes in the Au(111) surface, the local SAM environment around the nanografted region was found to vary. The energy exerted in the grafting process likely contributes to this local reconfiguration of the film, allowing local domains of the film near the grafted regions to settle into lower energy packing phases. The striped structure of these phases near the edge of the grafted region reasonably resemble the $6 \times \sqrt{3}$ phase of an alkanethiol, purportedly the lowest energy phase.[343] Interestingly, these rearrangements are only observed along the slow axis (top and bottom), and on the leading

edge of the graft (right), while only the $\left(\sqrt{3}\times\sqrt{3}\right)\mathrm{R}30°$ structure is observed on the trailing edge. The local variations in SAM structure at the boundaries of the graft would be expected to have minimal impact, however, as they all appear to be standing, well-ordered phases. The drop in height upon entering the grafted area is relatively sharp as well, indicating that the graft is uniformly surrounded by a well-ordered matrix that provides a consistent local environment.

6.3 Fabrication of Porphyrin Ensembles and Their Electronic Properties

With optimized parameters for nanografting, zinc porphyrin thiol ensembles were fabricated to investigate the relationship between ensemble size and conductivity. AFM and STM topography images are presented in Figure 6.11, with correlation in the structural features of the surfaces indicated. A CRLS ratio of ~2.0 was employed throughout to graft the porphyrin ensembles. The necessary load was determined by progressively nanoshaving the SAM with greater loads until a well-defined well was generated in the matrix SAM, yielding a grafting load of 45 nN, corresponding to a peak contact pressure of ~15 GPa. The features visible in Figure 6.11 correspond to square grafted features with dimensions ranging from 5-50 nm, though the actual size of the features varies due to instrument drift during the grafting process.

Figure 6.12 shows higher resolution images of individual grafts with dimensions measured directly. The internal structure of these grafts is not discernable, likely because the relatively weak interactions between the porphyrin molecules allow for considerable thermal fluctuation. Correlation between size of the feature and apparent height appears to exist, with smaller grafts on the order of 10-15 nm wide exhibiting apparent heights

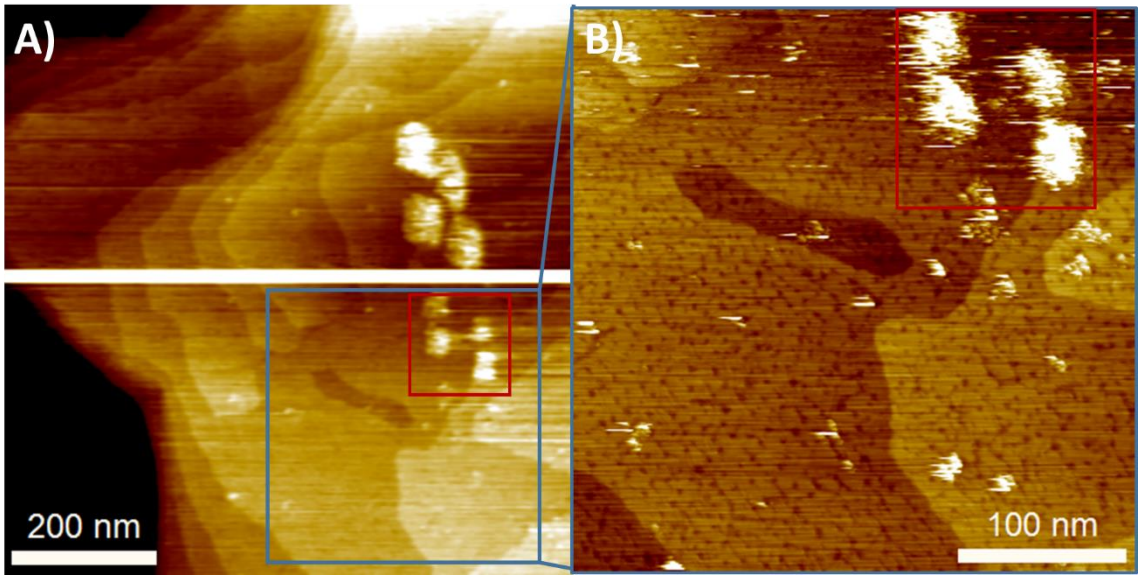


Figure 6.11. AFM topograph (A) and STM topograph (B) of nanografted zinc porphyrin thiol ensembles. Inside the red box are 25 nm nanografted features, and scattered through the image are 5 nm porphyrin ensembles. AFM Imaging condition: 0.5 nN. STM Imaging Conditions: Bias 1.4 V; Current 20 pA.

ranging from 0.5-1.0 nm. The variability in the apparent height may be due to variations in their internal architecture, since at this size the structures appear fairly irregular. For the largest features examined a consistent apparent height greater than 1 nm was observed, consistent with more efficient charge transport throughout the cluster.

To better understand the variation in charge transport mechanism from tunneling to sequential charge hopping with increasing aggregate size, it is important to consider how the charging energy of the porphyrin molecules and ensembles influences their conductance. The energy level diagram of the porphyrin adsorbed to the Au substrate is depicted in Figure 6.13. DFT electronic structure calculations and crossed-wire tunnel junction experiments indicate that the chemical potential of the porphyrin neutral state lies

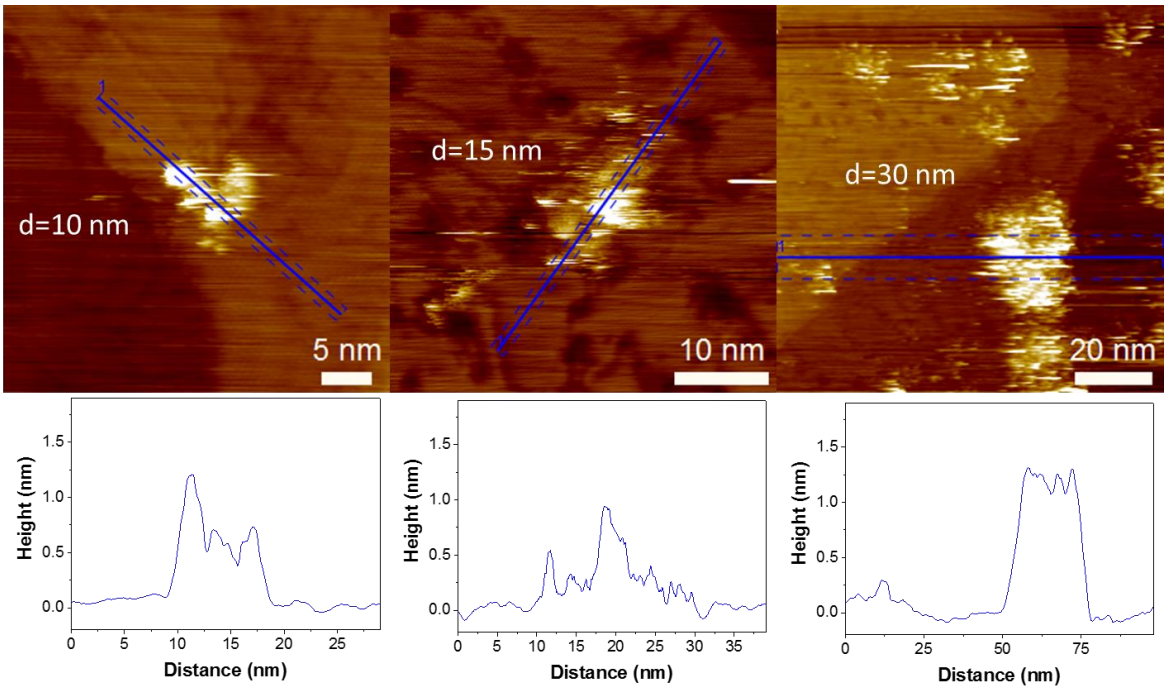


Figure 6.12. Nanografted structures of various dimensions (top) with corresponding line profiles (bottom). The smaller grafts exhibit heights ranging from 0.5-1.0 nm, suggesting enhanced conductivity relative to the single molecules, while the largest graft exhibits a uniform apparent height of ~1.2 nm.

close to the Fermi level. The energy required to introduce a hole, corresponding to removing one electron from the porphyrin HOMO, is approximately the charging energy. As the porphyrins are driven to aggregate, this positive charge can effectively be delocalized across the cluster, corresponding to reductions in the charging energy that render these hole transport pathways viable at the biases employed during imaging and spectroscopy of the porphyrin molecules on the surface.

The charging energy is directly related to the capacitance of the porphyrin islands. While this is not easily measured, it can be estimated using simple geometric models. A

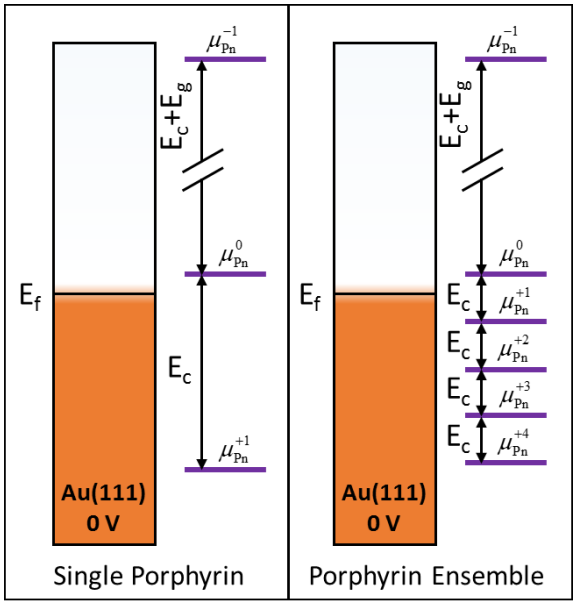


Figure 6.13. Energy level diagram of the porphyrin molecules on the Au(111) surface with no applied bias. For a single molecule, the bias required to align the Fermi level with cationic state is equivalent to the charge confinement energy. By forming ensembles of the porphyrin molecules, this charging energy is reduced, so that hole conduction pathways are more readily accessible.

sphere-in-sphere geometry, for example, has been employed to estimate the charging energy of a fullerene molecule tethered to an Au surface.[84] In this case, a reasonable geometry is a cylinder-in-cylinder configuration, corresponding to a row of porphyrin macrocycles pi-stacked and separated from the surface by their hydrocarbon tethers. The capacitance of such a configuration is:

$$C = L\frac{2\pi k\varepsilon_0}{\ln\left(b/a\right)} \quad \text{.....(6.6)}$$

Where L is the length of the cylindrical rods, k is the dielectric constant of the material separating the inner and outer cylinder, ε_0 the permittivity of vacuum, b the inner radius

of the outer cylinder, and a the outer radius of the inner cylinder. The dielectric consists of the material between the porphyrin macrocycles and the electrode, which consists of some combination of vacuum and hydrocarbon material, with a dielectric constant of 1-3. The outer radius of the inner cylinder, a, corresponds approximately to the diameter of the frontier states of the porphyrin molecules; as this is where initial charge storage will occur. From the DFT optimized structure, the inner radius is ~7 Å, measured from the zinc cation to a meso-pyridyl nitrogen. The inner radius of the outer cylinder, b, corresponds to the distance from the zinc cation to the electrode surface. Assuming a nominal tilt of the porphyrin molecule of 40°, b, is 13 Å. If an entirely pi-stacked structure is assumed, then L corresponds to the number of molecules times the molecular spacing of approximately 0.5 Å. From the capacitance, the charging energy can be determined by the equation:

$$E_C = \frac{0.5e^2}{C} \quad \text{.....(6.7)}$$

The charging energy as a function of the number of molecules in a porphyrin ensemble using this geometric model is depicted in Figure 6.14. It would appear that much of the reduction in charging energy comes from the addition of just a few molecules. The results on the other hand suggest that relatively large clusters on the order of tens of molecules are required to result in high transport efficiency consistent with charge hopping transport. The charging energy calculations employed here assume that the charge states are completely delocalized both spatially and temporally within the structure. This assumption is effectively true for metallic systems where electronic states are completely delocalized. However in these porphyrin complexes, the charge states are still confined to

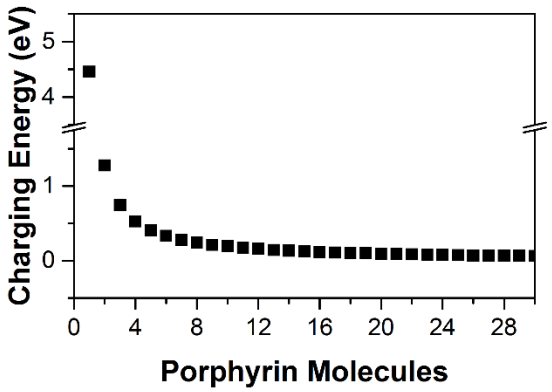


Figure 6.14. Charging energy of porphyrin islands using a cylinder-in-cylinder geometry to mimic pi-stacked porphyrin ensembles on the Au(111) surface.

the porphyrin macrocycles themselves, so that the total volume of the cluster is likely an overestimate of the total charge delocalization. Furthermore, electronic structure calculations of porphyrin dimers indicate little mixing of the frontier states,[305] which will limit the mobility of charges in the islands, further reducing the stabilizing benefit of charge delocalization. Finally, stabilization of charge afforded by the nearby electrode is overestimated as it does not wrap around the porphyrin cluster, and the model does not account for the STM tip which is also capacitively coupled to the surface and the porphyrin clusters. As a result, the general trend depicted in Figure 6.14 is reasonable, but the reduction in charging energy is more gradual owing to the overestimation of charge delocalization with increasing cluster size. This is supported by the results shown here, in which the onset of highly conductive pathways characteristic of sequential charge hopping occurs for molecular islands on the order of 10 nm.

6.4 Summary and Conclusion

To date, the nanografting technique has been demonstrated as a method for modifying the local chemistry of surfaces and as a means of patterning features onto a surface. The work presented herein demonstrates its effectiveness as a tool for managing charge transport through molecules at interfaces. By forming ensembles of a zinc porphyrin thiol on the Au(111) surface, it was demonstrated that controlled formation of molecular aggregates promotes a transition from tunneling transport to charge hopping based transport. This change in mechanism enhances the correlation between chemical structure and charge transport characteristics, allowing for greater control of current flow across interfaces. In addition, control of the chemical potential of the available transport pathways can be achieved by controlling the size and geometry of these aggregates, offering further control over their charge transport characteristics.

In addition, the sample transfer methods developed in this work offer the first glimpse into the detailed structure of nanografted features. It was found that the nanografting technique has limited impact on the underlying surface, with reconfiguration of the etch pits being the most obvious modification, and which is likely unavoidable on the Au(111) surface. Furthermore, at sufficiently high CRLS ratios employed in the formation of solid structures, complete clearance and replacement of the SAM matrix was observed. This indicates that the nanografting approach is viable as a means of creating well-ordered, chemically uniform structures on surfaces, which further supports its application in the formation of electronically active molecular components and features on surfaces for the control of charge transport at interfaces.

CHAPTER VII

CONCLUSION AND OUTLOOK

7.1 Summary

The effects of radial nanoconfinement on the mechanical properties of self-assembled monolayers, and the resulting changes in dissipative potential and surface passivation were investigated. It was found that, contrary to spectroscopic studies,[33] curvature has little direct effect on the disorder of alkylsilane films on nanoasperity surfaces, but rather the low coverage density of silane derived films on asperity surfaces is largely responsible for the observed reduction in film order. This results in lower dissipative potential, because fewer defects can form as a result of shear, and the large population of defects, which indicates little energy difference between the ordered and defect states, indicates that they are not effective dissipation pathways. Additionally, and perhaps more importantly, this disparity in film quality dramatically increased the exposure of the underlying surface to tribochemical interaction with an opposing surface, which will favor dissipative pathways at the reactive silica-silica interface that render reversible film dissipation pathways moot, and which will result in film and surface degradation.

Because surface coverage was found to be the driving factor in the protective benefit and dissipative potential of the film, the contact mechanics of the SAM protected contacts as a function of packing density were evaluated to determine how critical this matter of surface coverage actually is. It was found that for low coverage density asperity-asperity interactions the interactions were quite similar to a completely unfunctionalized contact,

with similar contact area and substantial pressure exerted at the silica-silica interface. Increasing the packing density to saturation revealed marked improvements in the protection of the interface, resulting in a larger area of contact and substantially lower direct interaction, and this was more apparent in asperity-flat interactions. With increasing coverage and decreasing contact curvature, at loads relevant to asperity contacts observed in devices, this picture was found to improve substantially, with contacts that exhibit lower curvature and full monolayer protection demonstrating very little interaction between the underlying substrates.

These contact simulations, and corresponding measurements of pressure and strain distributions, also used to explore the multi-regime friction response of SAMs, which provides insight into the function of boundary lubricants.[38,281] Using an asperity-flat model to simulate the AFM experiments, it was found that up to a threshold load, the strain distribution in the SAMs was uniform independent of load. Over this threshold, a clear increase in the film strain at the center of the contact was observed, strain that was comparable to the bond energies of the SAM, confirming that the source of increasing friction coefficient at greater loads corresponds to tribochemical pathways of film wear.

Geometric nanoconfinement of a hydrocarbon tethered zinc porphyrin thiol was also employed to examine how confinement could be used to influence the mode of transport through a molecular junction. The porphyrin macrocycle, decoupled from the electrode by a hydrocarbon tether, was found to exhibit tunneling characteristics when single, isolated molecules were probed. Through the development of pattern transfer techniques, the AFM nanografting technique was optimized to facilitate the fabrication of nanoscale

porphyrin islands, and subsequent grafting of the porphyrin molecules onto the Au(111) surface indicates a relationship between island size and conductivity. This change in conductivity, correlated to a change in transport mechanism in previous studies, indicates that the mechanism of molecular charge transport may be influenced by using nearest-neighbor interactions to build multi-molecule Coulomb islands via the tailoring of nearest-neighbor interactions. Furthermore, this size dependence in the clusters ability to support charge provides a bridge between the notions of single molecule conductance and thin film or conductive polymer conductance, wherein charge hopping between domains is supported by charge delocalization in the molecular or polymeric aggregates.

7.2 Outlook

7.2.1 Friction in Asperity-Asperity Contacts

Not only is MD simulation an effective tool in the study of contact mechanics, it has been used extensively to understand sliding friction. Friction of an asperity-asperity contact, however, is much more challenging owing to the difficulty of achieving sufficient signal-to-noise ratios due to short simulation timescales and the brevity of contact were the asperities to be sheared against one another. This, however, could be alleviated to some extent by measuring friction for rotating asperities. This will naturally neglect some of the mechanical details of shear that arise when two asperities slide against one another, providing instead a surface-focused evaluation of the frictional forces, rendering it ideal for studying the effects of a boundary lubricant.

This does present some technical challenge however. In MD friction studies, a restoring force is simulated, representing the bulk restoring forces of a sliding material or

the force exerted on an AFM tip by the cantilever as it slides along the surface. For a rotating surface, this would need to be replaced by a restoring torque. Development of methods to achieve this type of restoring action would allow these simulation and contact analysis techniques to be applied to "sliding" contacts, and could additionally be used to examine shear force and strain distributions at boundary lubricated contacts.

7.2.2 Alternative Boundary Lubricated Systems

While SAMs are an excellent model system for a boundary lubricated contact, in the majority of contacts, SAMs are ineffective owing to their lack of ability to heal or form *in situ* within the contact. Extending these simulations to include tribofilm materials, including layered materials like graphite, MoS_2, and hexagonal boron nitride would provide insights into how these layered materials can effectively lubricate an asperity contact. Graphite is of particular interest, as with the discovery of graphene it is possible for researchers to explore the crucial substrate-lubricant interactions of layered materials. An interesting phenomenon observed for layered materials is a dependence of the friction response on the number of layers within the contact,[344] the morphology of the surfaces in contact,[345] and the strength of the interactions between the substrate and graphene.[346] These issues have been explored extensively on flat surfaces, but less is known regarding the influence of surface curvature, which introduces strain and issues of conformity for the layered material at the interface.

MD simulation provides an excellent avenue to study these effects, and the asperity-asperity simulation contacts employed in this work can be adapted to consider these effects, in particular the role of local surface curvature and substrate-material interactions.

Through tuning of surface curvature by varying the size of the asperities, as well as varying the interactions between the graphene and the surface, it would be possible to explore these effects in atomic detail, providing a better understanding of the relationship between these various properties and the friction response of graphene.

7.2.3 Improvements in Nanografted Structure Architecture

While the studies of two-dimensional nanoconfinement of porphyrin thiols on a surface do indicate a correlation between structure size and conductivity, the internal architecture of these aggregates could not be verified and indeed, the images suggest considerable variability in quality and structural uniformity. Much of this could potentially be alleviated through better control of the forces employed during the nanografting process, which are inherently limited because the porphyrin thiols are grafted from a solution of DCM. Modification were made to the instrument to allow for this, but the imaging quality, and therefore the control over the nanografting process, were nevertheless limited. Substantial improvement could potentially be achieved by forming the structures via a two-step process, wherein the tether, absent the macrocycle, is first grafted onto the surface, followed by the attachment of the porphyrin. The principles of "click" chemistry were employed in the design of these porphyrin molecules, specifically facilitating the attachment of the tether to the macrocycle under gentle conditions, and thus it is reasonable to expect that the reaction could be driven to occur on the surface.

This approach yields two key benefits: first, the tether can be nanografted in ethanol, which is a much more suitable environment for AFM and can therefore be done with relative ease; second, by separating the design of the porphyrin macrocycle and its directed

assembly on the surface, a modular approach is achieved that facilitate synthetic design and tailoring of the porphyrin macrocycle, focused specifically on modifying electronic characteristics and nearest neighbor interactions.

REFERENCES

1. Leather, A. Intel Unveils 2014 Roadmap: 4 Fantastic New Processors For PC Enthusiasts. http://www.forbes.com/sites/antonyleather/2014/03/19/intel-unveils-2014-roadmap-4-fantastic-new-processors-for-pc-enthusiasts/ (accessed 06/24/14).

2. Krim, J. *Adv. Phys.* **2012,** 61, 155-323.

3. Persson, B. J., Introduction. In *Sliding Friction*, Springer Berlin Heidelberg: 1998; pp 1-8.

4. Bowden, F. P.; Tabor, D.; Palmer, F. *Am. J. Phys.* **1951,** 19, 428-429.

5. Greenwood, J. A.; Williamson, J. B. P. *Proc. R. Soc. London, Ser. A.* **1966,** 295, 300-319.

6. Bush, A. W.; Gibson, R. D.; Thomas, T. R. *Wear.* **1975,** 35, 87-111.

7. Barabási, A.-L.; Stanley., H. E., *Fractal Concepts in Surface Growth*. Cambridge University Press: 1995.

8. David, R.; Neumann, A. W. *Langmuir.* **2013,** 29, 4551-4558.

9. Persson, B. N. J. *Surf. Sci. Rep.* **2006,** 61, 201-227.

10. Persson, B. N. J. *Tribol. Lett.* **2014,** 54, 99-106.

11. Luan, B.; Robbins, M. *Tribol. Lett.* **2009,** 36, 1-16.

12. Hertz, H. J. *J. Reine Angew. Math.* **1881,** 92, 156-171.

13. Johnson, K. L., *Contact Mechanics*. Cambridge University Press.

14. Derjaguin, B. V.; Muller, V. M.; Toporov, Y. P. *J. Colloid Interface Sci.* **1975,** 53, 314-326.

15. Johnson, K. L.; Kendall, K.; Roberts, A. D. *Proc. R. Soc. London, Ser. A.* **1971,** 324, 301-313.

16. Carpick, R. W.; Agraït, N.; Ogletree, D. F.; Salmeron, M. *J. Vac. Sci. Technol., B.* **1996,** 14, 1289-1295.

17. Maugis, D. *J. Colloid Interface Sci.* **1992,** 150, 243-269.

18. Carpick, R. W.; Ogletree, D. F.; Salmeron, M. *J. Colloid Interface Sci.* **1999,** 211, 395-400.

19. Persson, B. J., Modern Experimental Methods and Results. In *Sliding Friction*, Springer Berlin Heidelberg: 1998; pp 17-35.

20. Persson, B. J., Sliding on Lubricated Surfaces. In *Sliding Friction*, Springer Berlin Heidelberg: 1998; pp 97-154.

21. Barthel, A. J.; Al-Azizi, A.; Surdyka, N. D.; Kim, S. H. *Langmuir*. **2013,** 30, 2977-2992.

22. Sánchez-López, J. C.; Erdemir, A.; Donnet, C.; Rojas, T. C. *Surf. Coat. Technol.* **2003,** 163–164, 444-450.

23. Gao, J.; Luedtke, W. D.; Gourdon, D.; Ruths, M.; Israelachvili, J. N.; Landman, U. *J. Phys. Chem. B.* **2004,** 108, 3410-3425.

24. Kim, S. H.; Asay, D. B.; Dugger, M. T. *Nano Today.* **2007,** 2, 22-29.

25. Asay, D.; Dugger, M.; Kim, S. *Tribol. Lett.* **2008,** 29, 67-74.

26. Eder, S.; Vernes, A.; Vorlaufer, G.; Betz, G. *J. Phys.: Condens. Matter.* **2011,** 23, 175004.

27. Vernes, A.; Eder, S.; Vorlaufer, G.; Betz, G. *Faraday Discuss.* **2012,** 156, 173-196.

28. Eder, S.; Vernes, A.; Betz, G. *Comput. Phys. Commun.* **2014,** 185, 217-228.

29. Chaudhury, M. K. *Curr. Opin. Colloid Interface Sci.* **1997,** 2, 65-69.

30. Bhushan, B., Self-Assembled Monolayers (SAMs) for Controlling Adhesion, Friction, and Wear. In *Springer Handbook of Nanotechnology*, Bhushan, B., Ed. Springer Berlin Heidelberg: 2007; pp 1379-1416.

31. Srinivasan, U.; Houston, M. R.; Howe, R. T.; Maboudian, R. *J. Microelectromech. Syst.* **1998,** 7, 252-260.

32. Feichtenschlager, B.; Lomoschitz, C. J.; Kickelbick, G. *J. Colloid Interface Sci.* **2011,** 360, 15-25.

33. Jones, R. L.; Pearsall, N. C.; Batteas, J. D. *J. Phys. Chem. C.* **2009,** 113, 4507-4514.

34. Naik, V. V.; Crobu, M.; Venkataraman, N. V.; Spencer, N. D. *J. Phys. Chem.*

Lett. **2013,** 4, 2745-2751.

35. Xiao, X.-D.; Liu, G.-y.; Charych, D. H.; Salmeron, M. *Langmuir.* **1995,** 11, 1600-1604.

36. Maboudian, R.; Ashurst, W. R.; Carraro, C. *Sens. Actuators, A.* **2000,** 82, 219-223.

37. Salmeron, M. *Tribol. Lett.* **2001,** 10, 69-79.

38. Xiao, X.; Hu, J.; Charych, D. H.; Salmeron, M. *Langmuir.* **1996,** 12, 235-237.

39. Colburn, T. J.; Leggett, G. J. *Langmuir.* **2007,** 23, 4959-4964.

40. Busuttil, K.; Geoghegan, M.; Hunter, C. A.; Leggett, G. J. *J. Am. Chem. Soc.* **2011,** 133, 8625-8632.

41. Nikogeorgos, N.; Hunter, C. A.; Leggett, G. J. *Langmuir.* **2012,** 28, 17709-17717.

42. Aviram, A.; Ratner, M. A. *Chem. Phys. Lett.* **1974,** 29, 277-283.

43. Nijhuis, C. A.; Reus, W. F.; Whitesides, G. M. *J. Am. Chem. Soc.* **2009,** 131, 17814-17827.

44. Leijnse, M.; Sun, W.; Nielsen, M. B.; Hedegård, P.; Flensberg, K. *J. Chem. Phys.* **2011,** 134, 104107.

45. Zhao, P.; Liu, D.-S.; Liu, H.-Y.; Li, S.-J.; Chen, G. *Org. Electron.* **2013,** 14, 1109-1115.

46. Zhou, J.; Samanta, S.; Guo, C.; Locklin, J.; Xu, B. *Nanoscale.* **2013,** 5, 5715-5719.

47. Diez-Perez, I.; Hihath, J.; Hines, T.; Wang, Z.-S.; Zhou, G.; Mullen, K.; Tao, N. *Nat. Nanotechnol.* **2011,** 6, 226-231.

48. Mishchenko, A.; Vonlanthen, D.; Meded, V.; Bürkle, M.; Li, C.; Pobelov, I. V.; Bagrets, A.; Viljas, J. K.; Pauly, F.; Evers, F., et al. *Nano Lett.* **2009,** 10, 156-163.

49. Venkataraman, L.; Klare, J. E.; Nuckolls, C.; Hybertsen, M. S.; Steigerwald, M. L. *Nature.* **2006,** 442, 904-907.

50. Basch, H.; Cohen, R.; Ratner, M. A. *Nano Lett.* **2005,** 5, 1668-1675.

51. Long, D. P.; Lazorcik, J. L.; Mantooth, B. A.; Moore, M. H.; Ratner, M. A.; Troisi, A.; Yao, Y.; Ciszek, J. W.; Tour, J. M.; Shashidhar, R. *Nat. Mater.* **2006,** 5, 901-908.

52. Piva, P. G.; DiLabio, G. A.; Pitters, J. L.; Zikovsky, J.; Rezeq, M. d.; Dogel, S.; Hofer, W. A.; Wolkow, R. A. *Nature.* **2005,** 435, 658-661.

53. Ulgut, B.; Abruña, H. D. *Chem. Rev.* **2008,** 108, 2721-2736.

54. Donley, C. L.; Blackstock, J. J.; Stickle, W. F.; Stewart, D. R.; Williams, R. S. *Langmuir.* **2007,** 23, 7620-7625.

55. Hsu, J. W. P. *Mater. Today.* **2005,** 8, 42-54.

56. Lau, C. N.; Stewart, D. R.; Bockrath, M.; Williams, R. S. *Appl. Phys. A.* **2005,** 80, 1373-1378.

57. Haick, H.; Cahen, D. *Acc. Chem. Res.* **2008,** 41, 359-366.

58. Chiechi, R. C.; Weiss, E. A.; Dickey, M. D.; Whitesides, G. M. *Angew. Chem. Int. Ed.* **2008,** 47, 142-144.

59. Kelley, T. W.; Granstrom, E.; Frisbie, C. D. *Adv. Mater.* **1999,** 11, 261-264.

60. Xu, B.; Tao, N. J. *Science.* **2003,** 301, 1221-1223.

61. Girard, C.; Joachim, C.; Chavy, C.; Sautet, P. *Surf. Sci.* **1993,** 282, 400-410.

62. Eigler, D. M.; Lutz, C. P.; Rudge, W. E. *Nature.* **1991,** 352, 600-603.

63. Krämer, S.; Fuierer, R. R.; Gorman, C. B. *Chem. Rev.* **2003,** 103, 4367-4418.

64. Fan, F.-R. F.; Yang, J.; Cai, L.; Price, D. W.; Dirk, S. M.; Kosynkin, D. V.; Yao, Y.; Rawlett, A. M.; Tour, J. M.; Bard, A. J. *J. Am. Chem. Soc.* **2002,** 124, 5550-5560.

65. Selzer, Y.; Cai, L.; Cabassi, M. A.; Yao, Y.; Tour, J. M.; Mayer, T. S.; Allara, D. L. *Nano Lett.* **2004,** 5, 61-65.

66. Weiss, E.; Wasielewski, M.; Ratner, M., Molecules as Wires: Molecule-Assisted Movement of Charge and Energy. In *Molecular Wires and Electronics*, Springer Berlin Heidelberg: 2005; Vol. 257, pp 103-133.

67. Coropceanu, V.; Cornil, J.; da Silva Filho, D. A.; Olivier, Y.; Silbey, R.; Brédas, J.-L. *Chem. Rev.* **2007,** 107, 926-952.

68. Klauk, H. *Chem. Soc. Rev.* **2010,** 39, 2643-2666.

69. Luo, L.; Choi, S. H.; Frisbie, C. D. *Chem. Mater.* **2010,** 23, 631-645.

70. Lu, Q.; Liu, K.; Zhang, H.; Du, Z.; Wang, X.; Wang, F. *ACS Nano.* **2009,** 3, 3861-3868.

71. Chen, C.-P.; Luo, W.-R.; Chen, C.-N.; Wu, S.-M.; Hsieh, S.; Chiang, C.-M.; Dong, T.-Y. *Langmuir.* **2013,** 29, 3106-3115.

72. Engelkes, V. B.; Beebe, J. M.; Frisbie, C. D. *J. Am. Chem. Soc.* **2004,** 126, 14287-14296.

73. Moth-Poulsen, K.; Bjornholm, T. *Nat. Nanotechnol.* **2009,** 4, 551-556.

74. Hines, T.; Diez-Perez, I.; Hihath, J.; Liu, H.; Wang, Z.-S.; Zhao, J.; Zhou, G.; Müllen, K.; Tao, N. *J. Am. Chem. Soc.* **2010,** 132, 11658-11664.

75. Choi, S. H.; Risko, C.; Delgado, M. C. R.; Kim, B.; Brédas, J.-L.; Frisbie, C. D. *J. Am. Chem. Soc.* **2010,** 132, 4358-4368.

76. Park, J.; Pasupathy, A. N.; Goldsmith, J. I.; Chang, C.; Yaish, Y.; Petta, J. R.; Rinkoski, M.; Sethna, J. P.; Abruna, H. D.; McEuen, P. L., et al. *Nature.* **2002,** 417, 722-725.

77. Kowalzik, P.; Atodiresei, N.; Gingras, M.; Caciuc, V.; Blügel, S.; Waser, R.; Karthäuser, S. *J. Phys. Chem. C.* **2011,** 115, 9204-9209.

78. Moth-Poulsen, K.; Patrone, L.; Stuhr-Hansen, N.; Christensen, J. B.; Bourgoin, J.-P.; Bjørnholm, T. *Nano Lett.* **2005,** 5, 783-785.

79. Yoon, H. J.; Shapiro, N. D.; Park, K. M.; Thuo, M. M.; Soh, S.; Whitesides, G. M. *Angew. Chem. Int. Ed.* **2012,** 51, 4658-4661.

80. Yoon, H. J.; Bowers, C. M.; Baghbanzadeh, M.; Whitesides, G. M. *J. Am. Chem. Soc.* **2013,** 136, 16-19.

81. Thuo, M. M.; Reus, W. F.; Simeone, F. C.; Kim, C.; Schulz, M. D.; Yoon, H. J.; Whitesides, G. M. *J. Am. Chem. Soc.* **2012,** 134, 10876-10884.

82. Nijhuis, C. A.; Reus, W. F.; Whitesides, G. M. *J. Am. Chem. Soc.* **2010,** 132, 18386-18401.

83. Lindsey, J. S.; Bocian, D. F. *Acc. Chem. Res.* **2011,** 44, 638-650.

84. Selzer, Y.; Allara, D. L. *Annu. Rev. Phys. Chem.* **2006,** 57, 593-623.

85. Yu, L. H.; Natelson, D. *Nano Lett.* **2003,** 4, 79-83.

86. Eom, K., *Simulations in Nanobiotechnology*. CRC Press: London, GBR, 2011.

87. Bichara, C.; Marsal, P.; Mottet, C.; Pellenq, R.; Ribeiro, F.; Saul, A.; Treglia, G.; Weissker, H. C. *Int. J. Nanotechnol.* **2012,** 9, 576-604.

88. Plimpton, S. *J. Comput. Phys.* **1995,** 117, 1-19.

89. Mitchon, L. N.; White, J. M. *Langmuir.* **2006,** 22, 6549-6554.

90. Brenner, D. W. *Phys. Rev. B: Condens. Matter.* **1990,** 42, 9458-9471.

91. van Duin, A. C. T.; Dasgupta, S.; Lorant, F.; Goddard, W. A. *J. Phys. Chem. A.* **2001,** 105, 9396-9409.

92. Voter, A. F. *Phys. Rev. B: Condens. Matter.* **1998,** 57, R13985-R13988.

93. Ryckaert, J.-P.; Bellemans, A. *Faraday Discuss.* **1978,** 66, 95-106.

94. Ryckaert, J.-P.; Ciccotti, G.; Berendsen, H. J. C. *J. Comput. Phys.* **1977,** 23, 327-341.

95. Baschnagel, J.; Binder, K.; Doruker, P.; Gusev, A.; Hahn, O.; Kremer, K.; Mattice, W.; Müller-Plathe, F.; Murat, M.; Paul, W., et al., Bridging the Gap Between Atomistic and Coarse-Grained Models of Polymers: Status and Perspectives. In *Viscoelasticity, Atomistic Models, Statistical Chemistry*, Springer Berlin Heidelberg: 2000; Vol. 152, pp 41-156.

96. Müller-Plathe, F. *ChemPhysChem.* **2002,** 3, 754-769.

97. Clementi, C. *Curr. Opin. Struct. Biol.* **2008,** 18, 10-15.

98. Murat, M.; Grest, G. S. *Macromolecules.* **1989,** 22, 4054-4059.

99. Wang, Y.; Jiang, W.; Yan, T.; Voth, G. A. *Acc. Chem. Res.* **2007,** 40, 1193-1199.

100. Barriga, J.; Coto, B.; Fernandez, B. *Tribol. Int.* **2007,** 40, 960-966.

101. Lorenz, C. D.; Tsige, M.; Rempe, S. B.; Chandross, M.; Stevens, M. J.; Grest, G. S. *J. Comput. Theor. Nanosci.* **2010,** 7, 2586-2601.

102. Guangtu, G.; Kevin Van, W.; Schall, J. D.; Judith, A. H. *J. Phys.: Condens. Matter.* **2006,** 18, S1737.

103. Luan, B.; Robbins, M. O. *Phys. Rev. E: Stat. Phys., Plasmas, Fluids,.* **2006,** 74,

026111.

104. Pei, Q. X.; Zhang, Y. W.; Shenoy, V. B. *Carbon.* **2010,** 48, 898-904.

105. Schapotschnikow, P.; Vlugt, T. J. H. *J. Phys. Chem. C.* **2010,** 114, 2531-2537.

106. Chandross, M.; Lorenz, C.; Grest, G.; Stevens, M.; Webb, E. *JOM.* **2005,** 57, 55-61.

107. Lorenz, C. D.; Chandross, M.; Lane, J. M. D.; Grest, G. S. *Modell. Simul. Mater. Sci. Eng.* **2010,** 18, 034005.

108. Ge, T.; Pierce, F.; Perahia, D.; Grest, G. S.; Robbins, M. O. *Phys. Rev. Lett.* **2013,** 110, 098301.

109. Allen, M. P.; Tildesley, D. J., *Computer Simulation of Liquids*. Oxford University Press: New York, NY, 1987.

110. Tuckerman, M. *J. Chem. Phys.* **1992,** 97, 1990.

111. Nosé, S. *J. Chem. Phys.* **1984,** 81, 511-519.

112. Schneider, T.; Stoll, E. *Phys. Rev. B: Condens. Matter.* **1978,** 17, 1302-1322.

113. Stillinger, F. H.; Weber, T. A. *Phys. Rev. B: Condens. Matter.* **1985,** 31, 5262-5271.

114. Tersoff, J. *Phys. Rev. B: Condens. Matter.* **1988,** 37, 6991-7000.

115. van Duin, A. C. T.; Strachan, A.; Stewman, S.; Zhang, Q.; Xu, X.; Goddard, W. A. *J. Phys. Chem. A.* **2003,** 107, 3803-3811.

116. Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. *J. Comput. Chem.* **1983,** 4, 187-217.

117. Ponder, J. W.; Case, D. A., Force Fields for Protein Simulations. In *Advances in Protein Chemistry*, Valerie, D., Ed. Academic Press: 2003; Vol. Volume 66, pp 27-85.

118. Jorgensen, W. L.; Maxwell, D. S.; Tirado-Rives, J. *J. Am. Chem. Soc.* **1996,** 118, 11225-11236.

119. Carré, A.; et al. *Europhys. Lett.* **2008,** 82, 17001.

120. van Beest, B. W. H.; Kramer, G. J.; van Santen, R. A. *Phys. Rev. Lett.* **1990,** 64, 1955-1958.

121. Makimura, D.; Metin, C.; Kabashima, T.; Matsuoka, T.; Nguyen, Q.; Miranda, C. *J. Mater. Sci.* **2010,** 45, 5084-5088.

122. Binnig, G.; Rohrer, H. *Surf. Sci.* **1983,** 126, 236-244.

123. Binnig, G.; Quate, C. F.; Gerber, C. *Phys. Rev. Lett.* **1986,** 56, 930-933.

124. Betzig, E.; Trautman, J. K. *Science.* **1992,** 257, 189-195.

125. Ewers, B. W.; Schuckman, A. E.; Batteas, J. D. *J. Chem. Educ.* **2014,** 91, 283-290.

126. Hallmark, V. M.; Chiang, S.; Meinhardt, K. P.; Hafner, K. *Phys. Rev. Lett.* **1993,** 70, 3740-3743.

127. Swart, I.; Sonnleitner, T.; Repp, J. *Nano Lett.* **2011,** 11, 1580-1584.

128. Stroscio, J. A.; Feenstra, R. M.; Fein, A. P. *Phys. Rev. Lett.* **1986,** 57, 2579-2582.

129. Stroscio, J. A.; Feenstra, R. M.; Newns, D. M.; Fein, A. P. *J. Vac. Sci. Technol., A.* **1988,** 6, 499-507.

130. Hipps, K. W., Tunneling Spectroscopy of Organic Monolayers and Single Molecules. In *Unimolecular and Supramolecular Electronics II*, Metzger, R. M., Ed. Springer Berlin Heidelberg: 2012; Vol. 313, pp 189-215.

131. Lennartz, M. C.; Baumert, M.; Karthäuser, S.; Albrecht, M.; Waser, R. *Langmuir.* **2011,** 27, 10312-10318.

132. Sader, J. E.; Sanelli, J. A.; Adamson, B. D.; Monty, J. P.; Wei, X.; Crawford, S. A.; Friend, J. R.; Marusic, I.; Mulvaney, P.; Bieske, E. J. *Rev. Sci. Instrum.* **2012,** 83, 103705.

133. Sader, J. E. *J. Appl. Phys.* **1998,** 84, 64-76.

134. Sheiko, S. S.; Möller, M.; Reuvekamp, E. M. C. M.; Zandbergen, H. W. *Phys. Rev. B: Condens. Matter.* **1993,** 48, 5675-5678.

135. Luis, G. R.; Jian, L. *J. Phys.: Condens. Matter.* **2009,** 21, 483001.

136. Xu, S.; Miller, S.; Laibinis, P. E.; Liu, G.-y. *Langmuir.* **1999,** 15, 7244-7251.

137. Lee, M. V.; Nelson, K. A.; Hutchins, L.; Becerril, H. A.; Cosby, S. T.; Blood, J. C.; Wheeler, D. R.; Davis, R. C.; Woolley, A. T.; Harb, J. N., et al. *Chem. Mater.* **2007,** 19, 5052-5054.

138. Torun, B.; Ozkaya, B.; Grundmeier, G. *Langmuir.* **2012,** 28, 6919-6927.

139. Liu, M.; Amro, N. A.; Liu, G.-y. *Annu. Rev. Phys. Chem.* **2008,** 59, 367-386.

140. Maboudian, R. *Surf. Sci. Rep.* **1998,** 30, 209-270.

141. Maboudian, R.; Carraro, C. *Annu. Rev. Phys. Chem.* **2004,** 55, 35-54.

142. Batteas, J.; Quan, X.; Weldon, M. *Tribol. Lett.* **1999,** 7, 121-128.

143. Binggeli, M.; Mate, C. M. *Appl. Phys. Lett.* **1994,** 65, 415-417.

144. Orlando, A., Birelle, J., Carlisle, J., Gerbi, J. E., Xiao, X., Peng, B., Espinosa, H. D. . *J. Phys.: Condens. Matter.* **2004,** 16, R539.

145. Luo, J. K.; Fu, Y. Q.; Le, H. R.; Williams, J. A.; Spearing, S. M.; Milne, W. I. *J. Micromech. Microeng.* **2007,** 17, S147.

146. Maboudian, R.; Ashurst, W. R.; Carraro, C. *Tribol. Lett.* **2002,** 12, 95-100.

147. Drobek, T.; Spencer, N. D. *Langmuir.* **2007,** 24, 1484-1488.

148. Yew, Y. K.; Minn, M.; Sinha, S. K.; Tan, V. B. C. *Langmuir.* **2011,** 27, 5891-5898.

149. Perry, S. S.; Yan, X.; Limpoco, F. T.; Lee, S.; Müller, M.; Spencer, N. D. *ACS Appl. Mater. Interfaces.* **2009,** 1, 1224-1230.

150. Limpoco, F. T.; Advincula, R. C.; Perry, S. S. *Langmuir.* **2007,** 23, 12196-12201.

151. Perkin, S.; Albrecht, T.; Klein, J. *PCCP.* **2010,** 12, 1243-1247.

152. Nainaparampil, J. J.; Eapen, K. C.; Sanders, J. H.; Voevodin, A. A. *J. Microelectromech. Syst.* **2007,** 16, 836-843.

153. Forbes, I. S.; Wilson, J. I. B. *Thin Solid Films.* **2002,** 420-421, 508-514.

154. Peiner, E.; Tibrewala, A.; Bandorf, R.; Lüthje, H.; Doering, L.; Limmer, W. *J. Micromech. Microeng.* **2007,** 17, S83.

155. Luo, X.; Yang, J.; Liu, H.; Wu, X.; Wang, Y.; Ma, Y.; Wei, S.-H.; Gong, X.; Xiang, H. *J. Am. Chem. Soc.* **2011,** 133, 16285-16290.

156. Tenne, R.; Homyonfer, M.; Feldman, Y. *Chem. Mater.* **1998,** 10, 3225-3238.

157. Scharf, T. W.; Prasad, S. V.; Dugger, M. T.; Kotula, P. G.; Goeke, R. S.; Grubbs,

R. K. *Acta Mater.* **2006,** 54, 4731-4743.

158. Santos, L. V.; Trava-Airoldi, V. J.; Iha, K.; Corat, E. J.; Salvadori, M. C. *Diamond Relat. Mater.* **2001,** 10, 1049-1052.

159. Zhao, X.; Perry, S. S. *ACS Appl. Mater. Interfaces.* **2010,** 2, 1444-1448.

160. Asay, D. B.; Dugger, M. T.; Ohlhausen, J. A.; Kim, S. H. *Langmuir.* **2007,** 24, 155-159.

161. Barnette, A. L.; Asay, D. B.; Ohlhausen, J. A.; Dugger, M. T.; Kim, S. H. *Langmuir.* **2010,** 26, 16299-16304.

162. Asay, D. B.; Hsiao, E.; Kim, S. H. *J. Appl. Phys.* **2011,** 110, 064326.

163. Chandross, M.; Lorenz, C. D.; Stevens, M. J.; Grest, G. S. *Langmuir.* **2008,** 24, 1240-1246.

164. Cheng, H.; Hu, Y. *Adv. Colloid Interface Sci.* **2012,** 171–172, 53-65.

165. Patton, S. T.; Cowan, W. D.; Eapen, K. C.; Zabinski, J. S. *Tribol. Lett.* **2001,** 9, 199-209.

166. Johnson, R. S.; Washburn, C. M.; Staton, A. W.; Moorman, M. W.; Manginell, R. P.; Dugger, M. T.; Dirk, S. M. *Macromol. Rapid Commun.* **2012,** 33, 1346-1350.

167. Jones, R. L.; Harrod, B. L.; Batteas, J. D. *Langmuir.* **2010,** 26, 16355-16361.

168. Knieling, T.; Lang, W.; Benecke, W. *Sens. Actuators, B.* **2007,** 126, 13-17.

169. Mayer, T. M.; de Boer, M. P.; Shinn, N. D.; Clews, P. J.; Michalske, T. A. *J. Vac. Sci. Technol., B.* **2000,** 18, 2433-2440.

170. Bhushan, B., Micro/Nanotribology of MEMS/NEMS Materials and Devices. In *Nanotribology and Nanomechanics*, Bhushan, B., Ed. Springer Berlin Heidelberg: 2005; pp 1031-1089.

171. Salmeron, M. *Tribol. Lett.* **2001,** 10, 69-79.

172. Tutein, A. B.; Stuart, S. J.; Harrison, J. A. *Langmuir.* **1999,** 16, 291-296.

173. Persson, B. N. J.; Albohr, O.; Tartaglino, U.; Volokitin, A. I.; Tosatti, E. *J. Phys.: Condens. Matter.* **2005,** 17, R1.

174. Gane, N.; Pfaelzer, P. F.; Tabor, D. *Proc. R. Soc. London, Ser. A.* **1974,** 340,

495-517.

175. Hook, D. A.; Timpe, S. J.; Dugger, M. T.; Krim, J. *J. Appl. Phys.* **2008,** 104, 034303.

176. Xu, C.; Jones, R. L.; Batteas, J. D. *Scanning.* **2008,** 30, 106-117.

177. Mo, Y.; Turner, K. T.; Szlufarska, I. *Nature.* **2009,** 457, 1116-1119.

178. de Boer, M. P.; Mayer, T. M. *MRS Bull.* **2001,** 26, 302-304.

179. Hyun, S.; Robbins, M. O. *Tribol. Int.* **2007,** 40, 1413-1422.

180. Campañá, C.; Müser, M. H.; Robbins, M. O. *J. Phys.: Condens. Matter.* **2008,** 20, 354013.

181. Cheng, S.; Robbins, M. *Tribol. Lett.* **2010,** 39, 329-348.

182. Luan, B.; Robbins, M. O. *Nature.* **2005,** 435, 929-932.

183. Luan, B. Q.; Hyun, S.; Molinari, J. F.; Bernstein, N.; Robbins, M. O. *Phys. Rev. E: Stat. Phys., Plasmas, Fluids,.* **2006,** 74, 046710.

184. Akarapu, S.; Sharp, T.; Robbins, M. O. *Phys. Rev. Lett.* **2011,** 106, 204301.

185. Yang, X.; Perry, S. S. *Langmuir.* **2003,** 19, 6135-6139.

186. Li, S.; Cao, P.; Colorado, R.; Yan, X.; Wenzl, I.; Shmakova, O. E.; Graupe, M.; Lee, T. R.; Perry, S. S. *Langmuir.* **2005,** 21, 933-936.

187. Szlufarska, I.; Chandross, M.; Carpick, R. W. *J. Phys. D: Appl. Phys.* **2008,** 41, 123001.

188. Jiménez, A.; Sarsa, A.; Blázquez, M.; Pineda, T. *J. Phys. Chem. C.* **2010,** 114, 21309-21314.

189. Ghorai, P. K.; Glotzer, S. C. *J. Phys. Chem. C.* **2007,** 111, 15857-15862.

190. Badia, A.; Singh, S.; Demers, L.; Cuccia, L.; Brown, G. R.; Lennox, R. B. *Chem. Eur. J.* **1996,** 2, 359-363.

191. Barrena, E.; Ocal, C.; Salmeron, M. *J. Chem. Phys.* **2000,** 113, 2413-2418.

192. Parikh, A. N.; Schivley, M. A.; Koo, E.; Seshadri, K.; Aurentz, D.; Mueller, K.; Allara, D. L. *J. Am. Chem. Soc.* **1997,** 119, 3135-3143.

193. Lee, S.; Shon, Y.-S.; Colorado, R.; Guenard, R. L.; Lee, T. R.; Perry, S. S. *Langmuir.* **2000,** 16, 2220-2224.

194. Gosvami, N. N.; Egberts, P.; Bennewitz, R. *J. Phys. Chem. A.* **2011,** 115, 6942-6947.

195. Soza, P.; Hansen, F. Y.; Taub, H.; Kiwi, M.; Cisternas, E.; Volkmann, U. G.; Campo, V. d. *Europhys. Lett.* **2011,** 95, 36001.

196. Barrena, E.; Ocal, C.; Salmeron, M. *Surf. Sci.* **2001,** 482–485, Part 2, 1216-1221.

197. Brewer, N. J.; Beake, B. D.; Leggett, G. J. *Langmuir.* **2001,** 17, 1970-1974.

198. Anand, M.; You, S.-S.; Hurst, K. M.; Saunders, S. R.; Kitchens, C. L.; Ashurst, W. R.; Roberts, C. B. *Ind. Eng. Chem. Res.* **2008,** 47, 553-559.

199. Park, J.; Aliaga, C.; Renzas, J.; Lee, H.; Somorjai, G. *Catal. Lett.* **2009,** 129, 1-6.

200. Aliaga, C.; Park, J. Y.; Yamada, Y.; Lee, H. S.; Tsung, C.-K.; Yang, P.; Somorjai, G. A. *J. Phys. Chem. C.* **2009,** 113, 6150-6155.

201. Tagliazucchi, M.; Tice, D. B.; Sweeney, C. M.; Morris-Cohen, A. J.; Weiss, E. A. *ACS Nano.* **2011,** 5, 9907-9917.

202. Zhang, L.; Wesley, K.; Jiang, S. *Langmuir.* **2001,** 17, 6275-6281.

203. Schall, J. D., Mikulski, P. T., Chateauneuf, G. M., Gao, G., Harrison, J. A., Molecular dynamics simulations of tribology. In *Superlubricity*, Ali, E.; Jean-Michel, M., Eds. Elsevier Science B.V.: Amsterdam, 2007; pp 79-102.

204. Mikulski, P. T.; Harrison, J. A. *J. Am. Chem. Soc.* **2001,** 123, 6873-6881.

205. Barry, P. R.; Chiu, P. Y.; Perry, S. S.; Sawyer, W. G.; Phillpot, S. R.; Sinnott, S. B. *Langmuir.* **2011,** 27, 9910-9919.

206. Harrison, J. A.; Gao, G. T.; Harrison, R. J.; Chateauneuf, G. M.; Mikulski, P. T. *Encyclopedia of Nanoscience and Nanotechnology.* **2004,** 3, 511-527.

207. Robbins, M. O.; Smith, E. D. *Langmuir.* **1996,** 12, 4543-4547.

208. Cheng, S.; Luan, B.; Robbins, M. O. *Phys. Rev. E: Stat. Phys., Plasmas, Fluids,.* **2010,** 81, 016102.

209. Chandross, M.; Webb, E. B., III; Stevens, M. J.; Grest, G. S.; Garofalini, S. H. *Phys. Rev. Lett.* **2004,** 93, 166103.

210. Lane, J. M. D.; Ismail, A. E.; Chandross, M.; Lorenz, C. D.; Grest, G. S. *Phys. Rev. E: Stat. Phys., Plasmas, Fluids,*. **2009,** 79, 050501.

211. Knippenberg, M. T., Mikulski, P. T., Harrison, J. A.,. *Modell. Simul. Mater. Sci. Eng.* **2010,** 18, 034002.

212. Lorenz, C. D.; Chandross, M.; Grest, G. S. *J. Adhes. Sci. Technol.* **2010,** 24, 2453-2469.

213. Humphrey, W.; Dalke, A.; Schulten, K. *J. Mol. Graphics Modell.* **1996,** 14, 33-38.

214. Lorenz, C. D.; Webb, E. B.; Stevens, M. J.; Chandross, M.; Grest, G. S. *Tribol. Lett.* **2005,** 19, 93-98.

215. Allara, D. L.; Parikh, A. N.; Judge, E. *J. Chem. Phys.* **1994,** 100, 1761-1764.

216. Kojio, K.; Ge, S.; Takahara, A.; Kajiyama, T. *Langmuir*. **1998,** 14, 971-974.

217. Fujii, M.; Sugisawa, S.; Fukada, K.; Kato, T.; Seimiya, T. *Langmuir*. **1995,** 11, 405-407.

218. Zhuravlev, L. T. *Langmuir*. **1987,** 3, 316-318.

219. Ducéré, J.-M.; Estève, A.; Dkhissi, A.; Rouhani, M. D.; Landa, G. *J. Phys. Chem. C.* **2009,** 113, 15652-15657.

220. Bierbaum, K.; Grunze, M.; Baski, A. A.; Chi, L. F.; Schrepp, W.; Fuchs, H. *Langmuir.* **1995,** 11, 2143-2150.

221. Carraro, C.; Yauw, O. W.; Sung, M. M.; Maboudian, R. *J. Phys. Chem. B.* **1998,** 102, 4441-4445.

222. Resch, R.; Grasserbauer, M.; Friedbacher, G.; Vallant, T.; Brunner, H.; Mayer, U.; Hoffmann, H. *Appl. Surf. Sci.* **1999,** 140, 168-175.

223. Glaser, A.; Foisner, J.; Hoffmann, H.; Friedbacher, G. *Langmuir.* **2004,** 20, 5599-5604.

224. Stevens, M. J. *Langmuir*. **1999,** 15, 2773-2778.

225. Tripp, C. P.; Hair, M. L. *Langmuir.* **1992,** 8, 1120-1126.

226. Barrena, E.; Kopta, S.; Ogletree, D. F.; Charych, D. H.; Salmeron, M. *Phys. Rev. Lett.* **1999,** 82, 2880-2883.

227. Chandross, M.; Grest, G. S.; Stevens, M. J.; III, E. B. W. Proc. SPIE 5343, Reliability, Testing, and Characterization of MEMS/MOEMS III, San Jose, CA, January 24, 2004; San Jose, CA, 2004; pp 207-214.

228. Lee, D. H.; Oh, T.; Cho, K. *J. Phys. Chem. B.* **2005,** 109, 11301-11306.

229. Nuzzo, R. G.; Korenic, E. M.; Dubois, L. H. *J. Chem. Phys.* **1990,** 93, 767-773.

230. Brown, D.; Clarke, J. H. R.; Okuda, M.; Yamazaki, T. *J. Chem. Phys.* **1994,** 100, 1684-1692.

231. MacPhail, R. A.; Strauss, H. L.; Snyder, R. G.; Elliger, C. A. *J. Phys. Chem.* **1984,** 88, 334-341.

232. Spori, D. M.; Venkataraman, N. V.; Tosatti, S. G. P.; Durmaz, F.; Spencer, N. D.; Zürcher, S. *Langmuir.* **2007,** 23, 8053-8060.

233. Porter, M. D.; Bright, T. B.; Allara, D. L.; Chidsey, C. E. D. *J. Am. Chem. Soc.* **1987,** 109, 3559-3568.

234. Hostetler, M. J.; Stokes, J. J.; Murray, R. W. *Langmuir.* **1996,** 12, 3604-3612.

235. Booth, B. D.; Vilt, S. G.; McCabe, C.; Jennings, G. K. *Langmuir.* **2009,** 25, 9995-10001.

236. Weeraman, C.; Yatawara, A. K.; Bordenyuk, A. N.; Benderskii, A. V. *J. Am. Chem. Soc.* **2006,** 128, 14244-14245.

237. Jackson, A. M.; Hu, Y.; Silva, P. J.; Stellacci, F. *J. Am. Chem. Soc.* **2006,** 128, 11135-11149.

238. Terrill, R. H.; Postlethwaite, T. A.; Chen, C.-h.; Poon, C.-D.; Terzis, A.; Chen, A.; Hutchison, J. E.; Clark, M. R.; Wignall, G. *J. Am. Chem. Soc.* **1995,** 117, 12537-12548.

239. Browne, K. P.; Grzybowski, B. A. *Langmuir.* **2010,** 27, 1246-1250.

240. Park, B.; Chandross, M.; Stevens, M. J.; Grest, G. S. *Langmuir.* **2003,** 19, 9239-9245.

241. Kudo, T.; Gordon, M. S. *J. Phys. Chem. A.* **2000,** 104, 4058-4063.

242. Kim, H.-J.; Kim, D.-E. *Nanoscale.* **2012,** 4, 3937-3944.

243. Mikulski, P.; Van Workum, K.; Chateaueuf, G.; Gao, G.; Schall, J.; Harrison, J. *Tribol. Lett.* **2011,** 42, 37-49.

244. Ulman, A. *Chem. Rev.* **1996,** 96, 1533-1554.

245. Ohtake, T.; Ogawa, K. *J. Dispersion Sci. Technol.* **2011,** 32, 407-414.

246. Wu, K.; Bailey, T. C.; Willson, C. G.; Ekerdt, J. G. *Langmuir*. **2005,** 21, 11795-11801.

247. Wang, R.; Wunder, S. L. *Langmuir.* **2000,** 16, 5008-5016.

248. Britt, D. W.; Hlady, V. *J. Colloid Interface Sci.* **1996,** 178, 775-784.

249. Wang, Y.; Lieberman, M. *Langmuir.* **2003,** 19, 1159-1167.

250. Bush, B. G.; DelRio, F. W.; Opatkiewicz, J.; Maboudian, R.; Carraro, C. *J. Phys. Chem. A.* **2007,** 111, 12339-12343.

251. Ito, Y.; Virkar, A. A.; Mannsfeld, S.; Oh, J. H.; Toney, M.; Locklin, J.; Bao, Z. *J. Am. Chem. Soc.* **2009,** 131, 9396-9404.

252. Major, R. C.; Zhu, X. Y. *Langmuir.* **2001,** 17, 5576-5580.

253. Carpick, R. W.; Salmeron, M. *Chem. Rev.* **1997,** 97, 1163-1194.

254. Park, J. Y.; Salmeron, M. *Chem. Rev.* **2013,** 114, 677-711.

255. Bowden, F. P.; Tabor, D., *The friction and lubrication of solids*. Claredon Press: Oxford, 1950.

256. Carbone, G.; Bottiglione, F. *J. Mech. Phys. Solids.* **2008,** 56, 2555-2572.

257. Tomlinson, G. A. *Philos. Mag.* **1929,** 7, 905-939.

258. Tomassone, M. S.; Sokoloff, J. B.; Widom, A.; Krim, J. *Phys. Rev. Lett.* **1997,** 79, 4798-4801.

259. Park, J. Y.; Qi, Y.; Ogletree, D. F.; Thiel, P. A.; Salmeron, M. *Phys. Rev. B: Condens. Matter.* **2007,** 76, 064108.

260. Priest, M.; Taylor, C. M. *Wear.* **2000,** 241, 193-203.

261. Barnes, A. M.; Bartle, K. D.; Thibon, V. R. A. *Tribol. Int.* **2001,** 34, 389-395.

262. Rymuza, Z. *Microsyst. Technol.* **1999,** 5, 173-180.

263. Marchetto, D.; Held, C.; Hausen, F.; Wählisch, F.; Dienwiebel, M.; Bennewitz, R. *Tribol. Lett.* **2012,** 48, 77-82.

264. Rapoport, L.; Fleischer, N.; Tenne, R. *J. Mater. Chem.* **2005,** 15, 1782-1788.

265. Bielecki, R.; Benetti, E.; Kumar, D.; Spencer, N. *Tribol. Lett.* **2012,** 45, 477-487.

266. Burris, D. L.; Sawyer, W. G. *Wear.* **2006,** 261, 410-418.

267. Bermúdez, M.-D.; Jiménez, A.-E.; Sanes, J.; Carrión, F.-J. *Molecules.* **2009,** 14, 2888-2908.

268. Koyama, M.; Hayakawa, J.; Onodera, T.; Ito, K.; Tsuboi, H.; Endou, A.; Kubo, M.; Del Carpio, C. A.; Miyamoto, A. *J. Phys. Chem. B.* **2006,** 110, 17507-17511.

269. Scharf, T. W.; Prasad, S. V. *J. Mater. Sci.* **2013,** 48, 511-531.

270. Kogut, L.; Komvopoulos, K. *J. Appl. Phys.* **2004,** 95, 576-585.

271. Enachescu, M.; Carpick, R. W.; Ogletree, D. F.; Salmeron, M. *J. Appl. Phys.* **2004,** 95, 7694-7700.

272. Reedy, E. D. *J. Mater. Res.* **2006,** 21, 2660-2668.

273. Lorenz, C. D.; Chandross, M.; Grest, G. S.; Stevens, M. J.; Webb, E. B. *Langmuir.* **2005,** 21, 11744-11748.

274. Chandross, M.; Lorenz, C. D.; Stevens, M. J.; Grest, G. S. *Langmuir.* **2008,** 24, 1240-1246.

275. Knippenberg, M. T.; Paul, T. M.; Judith, A. H. *Modell. Simul. Mater. Sci. Eng.* **2010,** 18, 034002.

276. Ye, Z.; Tang, C.; Dong, Y.; Martini, A. *J. Appl. Phys.* **2012,** 112, 116102.

277. Gao, G. T.; Mikulski, P. T.; Chateauneuf, G. M.; Harrison, J. A. *J. Phys. Chem. B.* **2003,** 107, 11082-11090.

278. Landman, U.; Luedtke, W. D.; Burnham, N. A.; Colton, R. J. *Science.* **1990,** 248, 454-461.

279. Yang, C.; Tartaglino, U.; Persson, B. N. J. *Eur. Phys. J. E: Soft Matter Biol. Phys.* **2006,** 19, 47-58.

280. Eder, S. J.; Vernes, A.; Betz, G. *Langmuir.* **2013,** 29, 13760-13772.

281. Flater, E. E.; Ashurst, W. R.; Carpick, R. W. *Langmuir.* **2007,** 23, 9242-9252.

282. Ewers, B. W.; Batteas, J. D. *J. Phys. Chem. C.* **2012,** 116, 25165-25177.

283. Lorenz, C. D.; Webb, E. B., III; Stevens, M. J.; Chandross, M.; Grest, G. S. *Tribol. Lett.* **2005,** 19, 93-98.

284. Judith, A. H.; Schall, J. D.; Knippenberg, M. T.; Guangtu, G.; Paul, T. M. *J. Phys.: Condens. Matter.* **2008,** 20, 354009.

285. Corana, A.; Marchesi, M.; Martini, C.; Ridella, S. *ACM Trans. Math. Softw.* **1987,** 13, 262-280.

286. Love, J. C.; Estroff, L. A.; Kriebel, J. K.; Nuzzo, R. G.; Whitesides, G. M. *Chem. Rev.* **2005,** 105, 1103-1170.

287. Nicosia, C.; Huskens, J. *Mater. Horiz.* **2014,** 1, 32-45.

288. Srinivasan, U.; Houston, M. R.; Howe, R. T.; Maboudian, R. *J. Microelectromech. Syst.* **1998,** 7, 252-260.

289. Patton, S.; Cowan, W.; Eapen, K.; Zabinski, J. *Tribol. Lett.* **2001,** 9, 199-209.

290. Ashurst, W. R.; Carraro, C.; Maboudian, R. *IEEE Trans. Device Mater. Reliab.* **2003,** 3, 173-178.

291. DePalma, V.; Tillman, N. *Langmuir.* **1989,** 5, 868-872.

292. Houston, J. E.; Kim, H. I. *Acc. Chem. Res.* **2002,** 35, 547-553.

293. Tabor, D.; Winterton, R. H. S. *Proc. R. Soc. London, Ser. A.* **1969,** 312, 435-450.

294. Joyce, S. A.; Houston, J. E. *Rev. Sci. Instrum.* **1991,** 62, 710-715.

295. Burns, A. R.; Houston, J. E.; Carpick, R. W.; Michalske, T. A. *Phys. Rev. Lett.* **1999,** 82, 1181-1184.

296. Liu, Y.; Evans, D. F.; Song, Q.; Grainger, D. W. *Langmuir.* **1996,** 12, 1235-1244.

297. Clear, S. C.; Nealey, P. F. *Langmuir.* **2001,** 17, 720-732.

298. Brewer, N. J.; Leggett, G. J. *Langmuir.* **2004,** 20, 4109-4115.

299. Mo, Y.; Szlufarska, I. *Phys. Rev. B: Condens. Matter.* **2010,** 81, 035405.

300. Ewers, B. W.; Batteas, J. D. *Langmuir.* **2014**. DOI: 10.1021/la500032f.

301. Chen, Y. L.; Helm, C. A.; Israelachvili, J. N. *J. Phys. Chem.* **1991,** 95, 10736-10747.

302. Joyce, S. A.; Thomas, R. C.; Houston, J. E.; Michalske, T. A.; Crooks, R. M. *Phys. Rev. Lett.* **1992,** 68, 2790-2793.

303. Jansen, L.; Lantz, M. A.; Knoll, A. W.; Schirmeisen, A.; Gotsmann, B. *Langmuir.* **2014,** 30, 1557-1565.

304. Tian, F.; Xiao, X.; Loy, M. M. T.; Wang, C.; Bai, C. *Langmuir.* **1998,** 15, 244-249.

305. Schuckman, A. E.; Ewers, B. W.; Lam, H. Y.; Tome, J. P. C.; Perez, L. M.; Drain, C. M.; Kushmerick, J. G.; Batteas, J. D. "Utilizing Nearest-Neighbor Interactions to Alter Charge Transport Mechanisms in Molecular Assemblies of Porphyrins on Surfaces". To be submitted to *J. Phys. Chem. C.*

306. Huang, Z.; Xu, B.; Chen, Y. L.; Ventra, M. D.; Tao, N. *Nano Lett.* **2006,** 6, 1240-1244.

307. Lu, W.; Lieber, C. M. *Nat. Mater.* **2007,** 6, 841-850.

308. Armstrong, N.; Hoft, R. C.; McDonagh, A.; Cortie, M. B.; Ford, M. J. *Nano Lett.* **2007,** 7, 3018-3022.

309. Donhauser, Z. J.; Mantooth, B. A.; Kelly, K. F.; Bumm, L. A.; Monnell, J. D.; Stapleton, J. J.; Price, D. W.; Rawlett, A. M.; Allara, D. L.; Tour, J. M., et al. *Science.* **2001,** 292, 2303-2307.

310. Feng, M.; Gao, L.; Du, S. X.; Deng, Z. T.; Cheng, Z. H.; Ji, W.; Zhang, D. Q.; Guo, X. F.; Lin, X.; Chi, L. F., et al. *Adv. Funct. Mater.* **2007,** 17, 770-776.

311. Seo, K.; Konchenko, A. V.; Lee, J.; Bang, G. S.; Lee, H. *J. Am. Chem. Soc.* **2008,** 130, 2553-2559.

312. Pathem, B. K.; Claridge, S. A.; Zheng, Y. B.; Weiss, P. S. *Annu. Rev. Phys. Chem.* **2013,** 64, 605-630.

313. Yaffe, O.; Scheres, L.; Puniredd, S. R.; Stein, N.; Biller, A.; Lavan, R. H.; Shpaisman, H.; Zuilhof, H.; Haick, H.; Cahen, D., et al. *Nano Lett.* **2009,** 9, 2390-2394.

314. Har-Lavan, R.; Yaffe, O.; Joshi, P.; Kazaz, R.; Cohen, H.; Cahen, D. *AIP Adv.* **2012,** 2, 012164.

315. Asbury, J. B.; Hao, E.; Wang, Y.; Lian, T. *J. Phys. Chem. B.* **2000,** 104, 11957-11964.

316. Weiss, E. A. *Acc. Chem. Res.* **2013,** 46, 2607-2615.

317. Reynal, A.; Forneli, A.; Martinez-Ferrero, E.; Sánchez-Díaz, A.; Vidal-Ferran, A.; O'Regan, B. C.; Palomares, E. *J. Am. Chem. Soc.* **2008,** 130, 13558-13567.

318. Kushmerick, J. G.; Holt, D. B.; Pollack, S. K.; Ratner, M. A.; Yang, J. C.; Schull, T. L.; Naciri, J.; Moore, M. H.; Shashidhar, R. *J. Am. Chem. Soc.* **2002,** 124, 10654-10655.

319. Alemani, M.; Peters, M. V.; Hecht, S.; Rieder, K.-H.; Moresco, F.; Grill, L. *J. Am. Chem. Soc.* **2006,** 128, 14446-14447.

320. He, J.; Fu, Q.; Lindsay, S.; Ciszek, J. W.; Tour, J. M. *J. Am. Chem. Soc.* **2006,** 128, 14828-14835.

321. Roth, K. M.; Dontha, N.; Dabke, R. B.; Gryko, D. T.; Clausen, C.; Lindsey, J. S.; Bocian, D. F.; Kuhr, W. G. *J. Vac. Sci. Technol., B.* **2000,** 18, 2359-2364.

322. Xiao, X.; Brune, D.; He, J.; Lindsay, S.; Gorman, C. B.; Tao, N. *Chem. Phys.* **2006,** 326, 138-143.

323. Wu, S. W.; Ogawa, N.; Nazin, G. V.; Ho, W. *J. Phys. Chem. C.* **2008,** 112, 5241-5244.

324. Stadler, R.; Geskin, V.; Cornil, J. *J. Phys.: Condens. Matter.* **2008,** 20, 374105.

325. Chan, Y.-H.; Schuckman, A. E.; Perez, L. M.; Vinodu, M.; Drain, C. M.; Batteas, J. D. *J. Phys. Chem. C.* **2008,** 112, 6110-6118.

326. Winters, M. U.; Dahlstedt, E.; Blades, H. E.; Wilson, C. J.; Frampton, M. J.; Anderson, H. L.; Albinsson, B. *J. Am. Chem. Soc.* **2007,** 129, 4291-4297.

327. Jurow, M.; Schuckman, A. E.; Batteas, J. D.; Drain, C. M. *Coord. Chem. Rev.* **2010,** 254, 2297-2310.

328. Cui, Y.; Wu, Y.; Lu, X.; Zhang, X.; Zhou, G.; Miapeh, F. B.; Zhu, W.; Wang, Z.-S. *Chem. Mater.* **2011,** 23, 4394-4401.

329. Parr, R. G.; Yang, W., *Density Functional Theory of Atoms and Molecules*. Oxford University Press: New York, 1989.

330. *Gaussian*, version 03 Revision 02; Gaussian, Inc.: Wallingford, CT, 2004.

331. Tao, J.; Perdew, J. P.; Staroverov, V. N.; Scuseria, G. E. *Phys. Rev. Lett.* **2003,** 91, 146401.

332. Petersson, G. A.; Bennett, A.; Tensfeldt, T. G.; Al-Laham, M. A.; Shirley, W. A.; Mantzaris, J. *J. Chem. Phys.* **1988,** 89, 2193-2218.

333. Dunning, T. H., Jr.; Hay, P. J., Gaussian Basis Sets for Molecular Calculations. In *Methods of Electronic Structure Theory*, Schaefer, H., III, Ed. Springer US: 1977; Vol. 3, pp 1-27.

334. Gorelsky, S. I.; Lever, A. B. P. *J. Organomet. Chem.* **2001,** 635, 187-196.

335. Bumm, L. A.; Arnold, J. J.; Dunbar, T. D.; Allara, D. L.; Weiss, P. S. *J. Phys. Chem. B.* **1999,** 103, 8122-8127.

336. Sedghi, G.; Sawada, K.; Esdaile, L. J.; Hoffmann, M.; Anderson, H. L.; Bethell, D.; Haiss, W.; Higgins, S. J.; Nichols, R. J. *J. Am. Chem. Soc.* **2008,** 130, 8582-8583.

337. Ngunjiri, J. N.; Kelley, A. T.; Lejeune, Z. M.; Li, J.-R.; Lewandowski, B. R.; Serem, W. K.; Daniels, S. L.; Lusker, K. L.; Garno, J. C. *Scanning.* **2008,** 30, 123-136.

338. Schlenoff, J. B.; Li, M.; Ly, H. *J. Am. Chem. Soc.* **1995,** 117, 12528-12536.

339. Kelley, A. T.; Ngunjiri, J. N.; Serem, W. K.; Lawrence, S. O.; Yu, J.-J.; Crowe, W. E.; Garno, J. C. *Langmuir.* **2010,** 26, 3040-3049.

340. Helt, J. M.; Batteas, J. D. *Langmuir.* **2006,** 22, 6130-6141.

341. te Riet, J.; Smit, T.; Coenen, M. J. J.; Gerritsen, J. W.; Cambi, A.; Elemans, J. A. A. W.; Speller, S.; Figdor, C. G. *Soft Matter.* **2010,** 6, 3450-3454.

342. Jun-Fu Liu, S. R. P.; Von Ehr, J. R.; Baur, C.; Stallcup, R.; Randall, J.; Bray, K. *Appl. Phys. Lett.* **2004,** 84, 1359-1361.

343. Noh, J.; Hara, M. *Langmuir.* **2002,** 18, 1953-1956.

344. Lee, C.; Li, Q.; Kalb, W.; Liu, X.-Z.; Berger, H.; Carpick, R. W.; Hone, J. *Science.* **2010,** 328, 76-80.

345. Yalin, D. *J. Phys. D: Appl. Phys.* **2014,** 47, 055305.

346. Li, Q.; Lee, C.; Carpick, R. W.; Hone, J. *Phys. Status Solidi B.* **2010,** 247, 2909-2914.

APPENDIX A

SUPPLEMENTAL TRAJECTORY INFORMATION AND RESULTS FOR SAM STRUCTURES ON SILICA SURFACES

A.1 Energy and Temperature Trajectories of Simulations

The simulations were performed with controlled temperature using a Langevin thermostat with a damping constant of 10 fs. Figure A.1 demonstrates the temperature control and total system energy observed for the dodecylsilane functionalized particles. The simulations began with an initial annealing step to decrease the correlation time, with the effect of minimizing initial condition effects in the relaxation process.

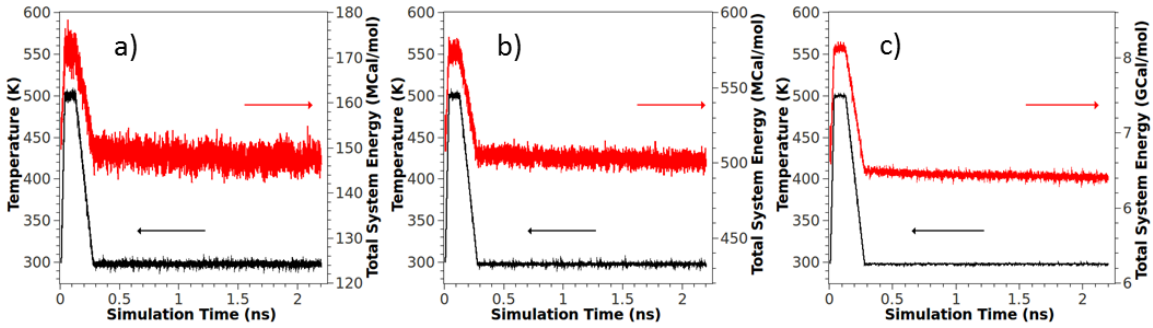


Figure A.1: Temperature and Energy trajectories for the simulations of dodecylsilane functionalized 7 nm (a), 12 nm (b), and 40 nm (c) particles, demonstrating stable temperatures and system energies.

A.2 Supplementary Data

Figure A.2 demonstrates the position dependence of *gauche* defect densities, demonstrating that the tendency for *gauche* defects to form near the particle surfaces is constant for all low density systems. For the flat systems of increasing coverage, the first position away from the surface still presents a high defect density, this is likely because of the unnatural way the molecules are bound to the surface, though the effect quickly

dissipates further down the chains.

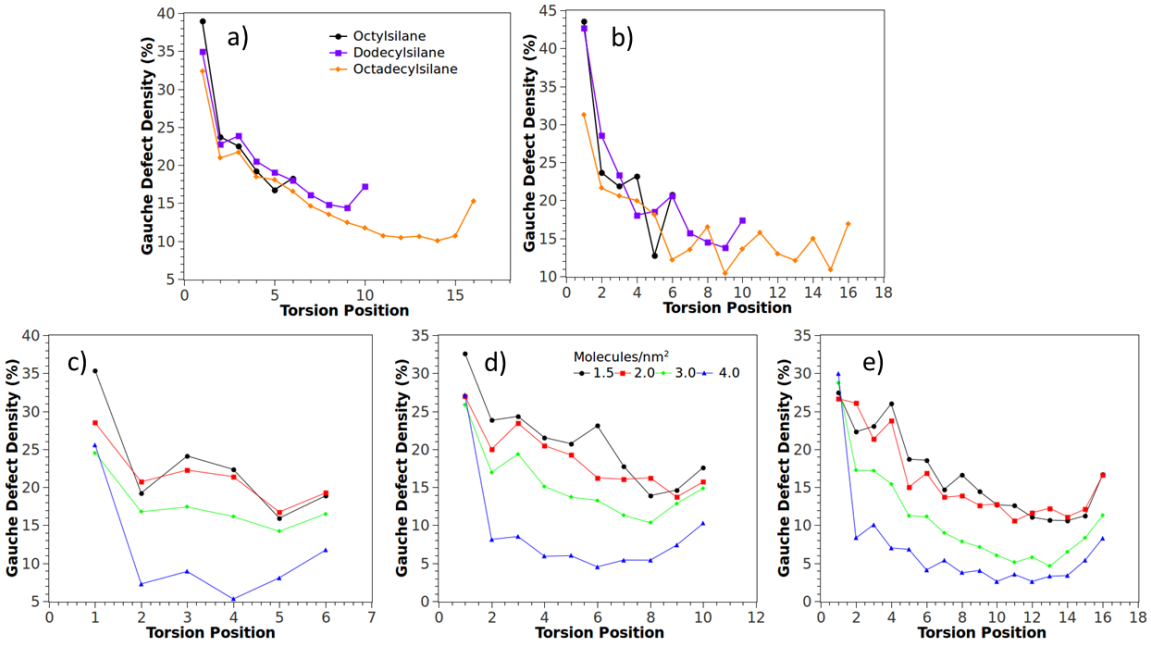


Figure A.2: *Gauche* defect densities as a function of location away from the surface, position 1 corresponding to the Si-C-C-C torsion closest to the surface and successive positions further up the chains. Results for the 7 nm (a) and 40 nm particles show the tendency for the majority of defects to form near the particle surface. As a function of packing density (c-e, C8,C12,C18 respectively), ordering appears to progress closer to the surface with increase packing density, with the highest packing density demonstrating almost uniform defect densities beyond the first torsion.

A.3 Range of Motion Analysis

The range-of-motion(ROM) of the molecules on the surface was performed employing a computation developed by Harrison and coworkers.[243] The following equation describes the computation employed here:

$$ROM = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{2}{M(M-1)}\sum_{n=1}^{M}\sum_{m=n+1}^{M}|\vec{r}_i(n)-\vec{r}_i(m)|\right)$$

Where N is the number of molecules considered, M is the number of snapshots

considered, and $r_i(n)$ is the vector connecting the anchor and tail of molecule i at snapshot n. ROM was calculated over intervals of 10 snapshots, and the value of N varied depending on the coverage and the radius of curvature, with larger particles having larger values of N.

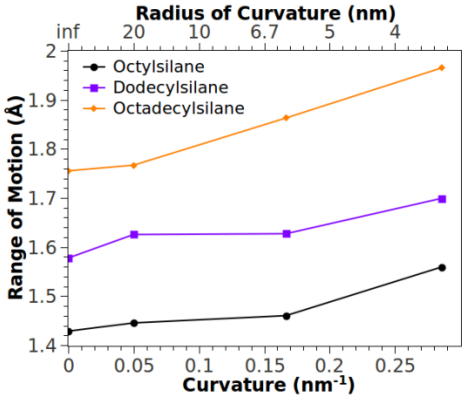


Figure A.3: Range-of-motion for the three chain lengths and curvatures studied. Increasing ROM with chain length correspond to increased degrees of freedom. With increased curvature, ROM also increases, suggesting a less rigid film that will produce greater friction.

Figure A.3 demonstrates the ROM as a function of curvature. The results here generally support the results already obtained, with greater curvature producing greater ROM, associated with the greater disorder in the film. As is the case on flat surfaces, ROM increases with chain length, as it is also dependent on the molecular degrees of freedom at low coverage, this trend vanishes at high coverage, where packing effects act to limit the ROM.

A.4 Additional Simulation Images

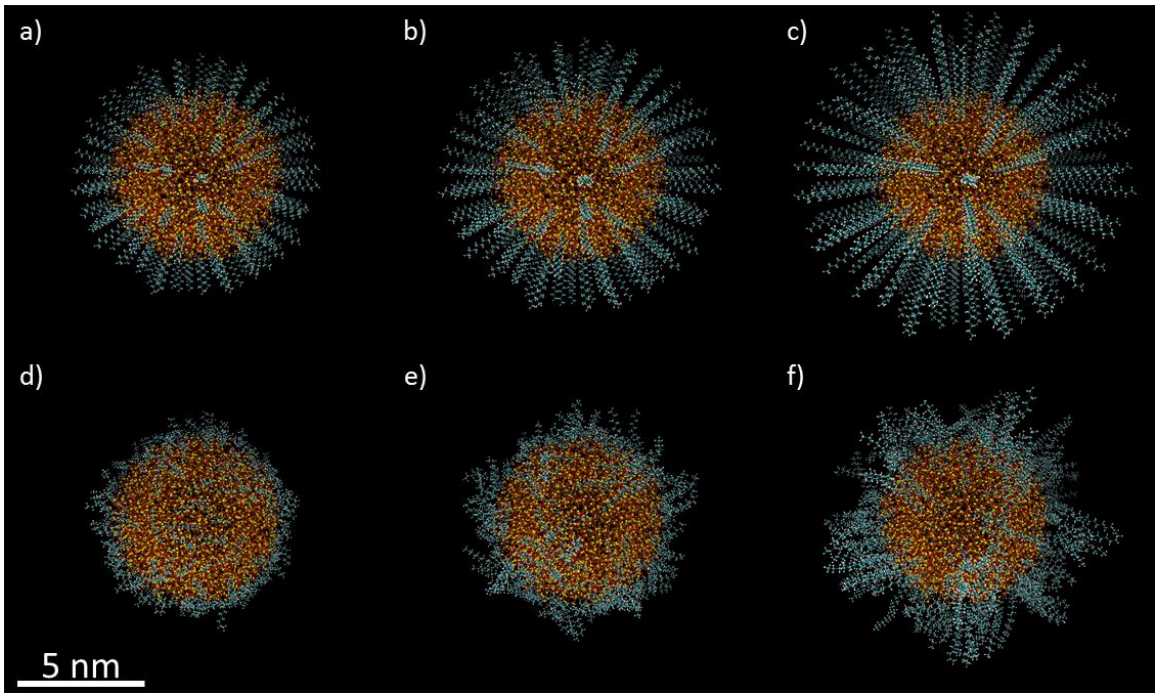


Figure A.4: 7 nm Particles, a-c before simulation, d-f after simulation, C8, C12, C18 respectively.

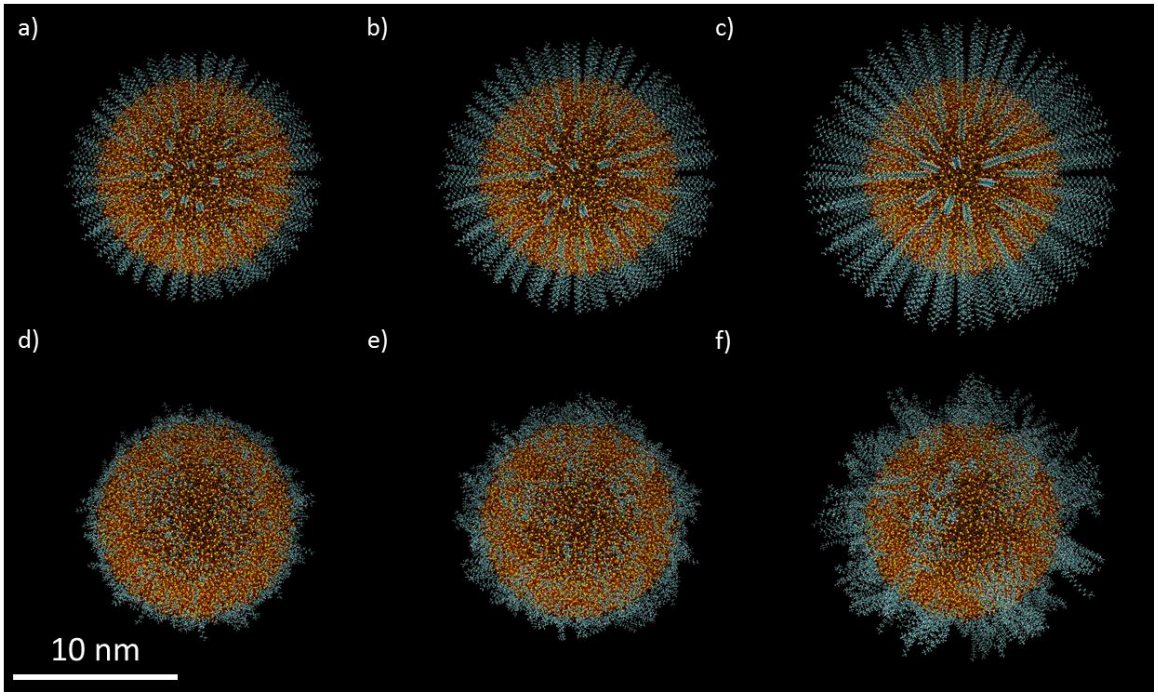


Figure A.5: 12 nm Particles, a-c before simulation, d-f after simulation, C8, C12, C18 respectively.

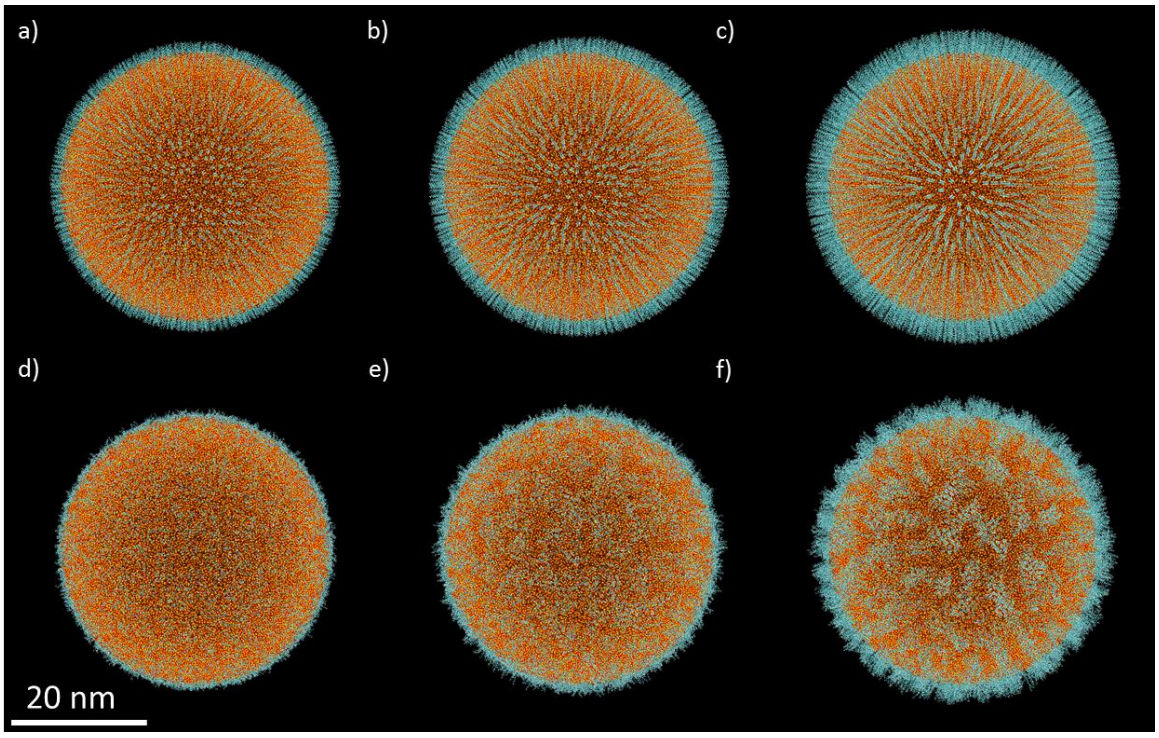


Figure A.6: 40 nm Particles, a-c before simulation, d-f after simulation, C8, C12, C18 respectively.

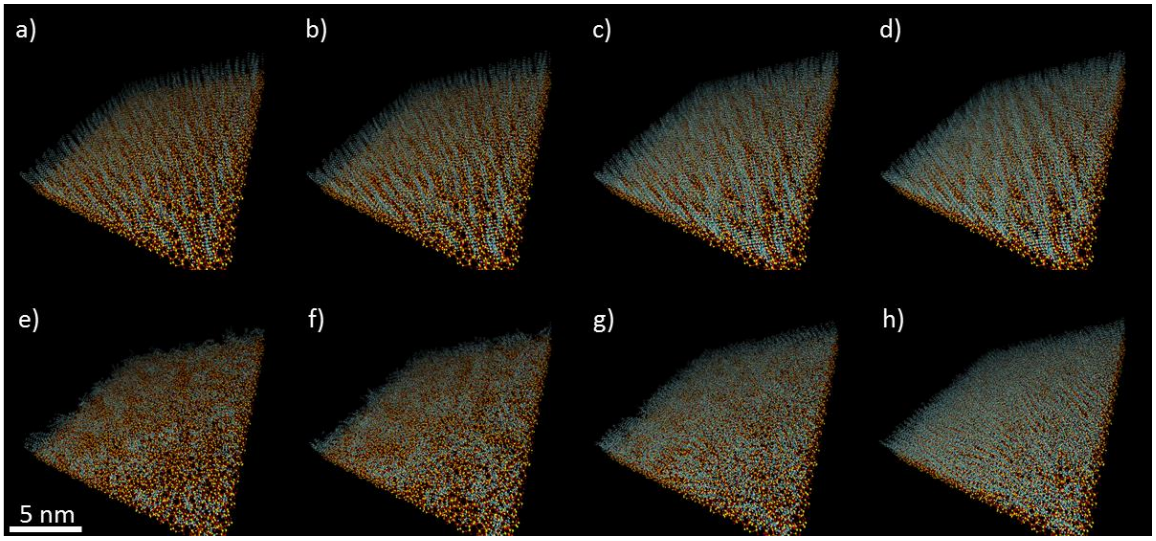


Figure A.7: Flat surface simulations of varied coverage, a-d before simulation, e-h after simulation, 1.5, 2, 3, and 4 molecules/nm^2 respectively. The functionalizing molecule in all cases is dodecylsilane.

APPENDIX B

SIMULATION MODEL PREPARATION AND ANALYSIS SOFTWARE

The functionalized surface models employed in this work were developed virtually from scratch. The software presented in this appendix was used to aid in preparing the models and analyzing the resulting simulated structures from snapshot data produced by the LAMMPS software package. C++ classes are used as containers, with the 'Particle' class containing class lists of types and elements, which are themselves classes, as depicted in Figure B.1.

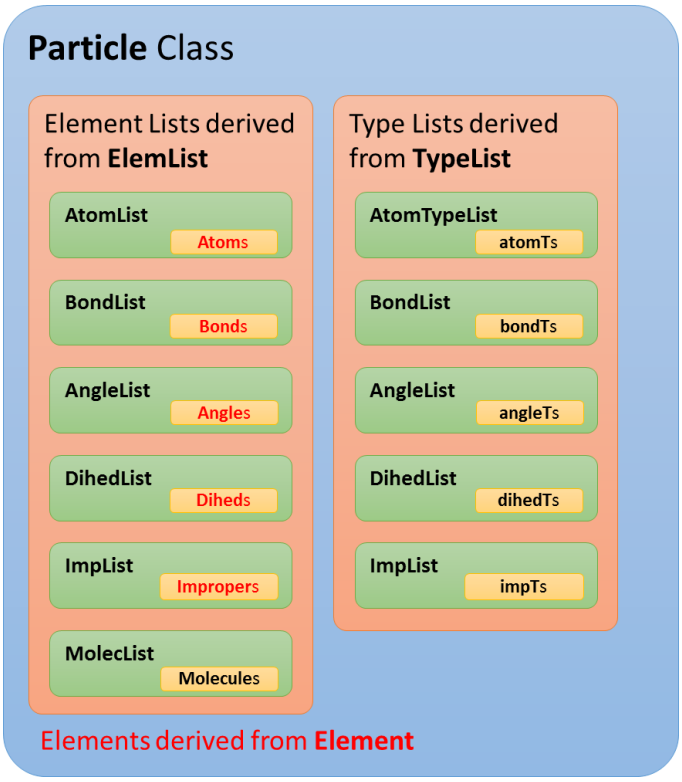


Figure B.1. Container hierarchy for libraries generating and analyzing functionalized particle surfaces. Particle is the overarching class, containing system details like boundary dimensions and numbers of snapshots, as well as lists of the various elements and types used in the system.

B.1 Primary include files and definitions

This include file is required for compilation of all subsequent files, covers all class definitions and basic standard includes.

inc.h

```
#ifndef INC_H_
#define INC_H_

//Constant definitions
#define PI 3.14159265358979323846264338327795
#define R2D 57.29577951308232087679815481410 5
#define DEB(x) cout << #x << endl;

//Intel performance primitives are used primarily for the mathematics.  Some functions were
    written to take the place of the IPP libraries assuming that they might become unavailable,
    but this is not always the case.  The following defines affect the compile time configuration
    specifying single or double floats
#ifdef ipp
    #include "ipps.h"
    #include "ippm.h"
    #include "ippvm.h"
    #ifdef F64
        #define FLOAT Ipp64f
        #define IPS 64f
        #define IPPF1(x) x ## _ ## 64f
        #define IPPF2(x,y) x ## _ ## 64f ## _ ## y
        #define IPPFIX(x) x ## _ ## 64f_A53
    #else
        #define FLOAT Ipp32f
        #define IPS 32f
        #define IPPF1(x) x ## _ ## 32f
        #define IPPF2(x,y) x ## _ ## 64f ## _ ## y
        #define IPPFIX(x) x## _ ## 32f_A24
    #endif
#else
    #ifdef F64
        #define FLOAT double
    #else
        #define FLOAT float
    #endif
#endif

//Primary class definitions
//Item Classes
class Atom;
class Bond;
class Angle;
class Dihed;
class Improper;
//Type Classes
class atomT;
class bondT;
class angleT;
class dihedT;
class impT;
//Element List Classes
class AtomList;
class BondList;
```

```
class AngleList;
class DihedList;
class ImpList;
//Type List Classes
class AtomTypeList;
class BondTypeList;
class AngleTypeList;
class DihTypeList;
class ImpTypeList;
//Primary class
class Particle;
//Molecule is a LAMMPS construct that is similarly preserved
class Molecule;
class MolecList;

//Standard library includes
#include <iostream>
#include <iomanip>
#include <map>
#include <queue>
#include <list>
#include <vector>
#include <deque>
#include <set>
#include <sstream>
#include <fstream>
using namespace std;
#include <stdlib.h>
#include <math.h>

#include "stdint.h"
#define __STDC_FORMAT_MACROS
#include "inttypes.h"

typedef int tagint;
typedef int64_t bigint;
#define BIGINT_FORMAT "%" PRId64

//Includes for the various class definitions
//Template classes for the lists
#include "TypeList.h"
#include "ElemList.h"
//Template class for items
#include "Elem.h"
//Definitions that correlate item masses to element names
#include "elements.h"
//Various class definitions
#include "Atom.h"
#include "Bond.h"
#include "Angle.h"
#include "Dihed.h"
#include "Molecule.h"
#include "Improper.h"
#include "atomT.h"
#include "bondT.h"
#include "angleT.h"
#include "dihedT.h"
#include "impT.h"
#include "AtomList.h"
#include "BondList.h"
#include "AngleList.h"
#include "DihedList.h"
#include "MolecList.h"
#include "ImpList.h"
#include "AtomTypeList.h"
```

```
#include "BondTypeList.h"
#include "AngleTypeList.h"
#include "DihTypeList.h"
#include "ImpTypeList.h"
#include "Particle.h"

//Non-class functions, some defined here, others elsewhere
void normalizeVecs(FLOAT*,int);
void relVecs(vector<Atom*>&, FLOAT*, int s=0, int d=0);
void relVecsF2R(vector<Atom*> atoms, FLOAT* out, int s=0, int d=0);
void radSpir(FLOAT* centers, FLOAT radius, int c, int N, FLOAT* out, FLOAT* normedOut);
int rectGrid(FLOAT* dims, FLOAT radius, FLOAT* heights, int dCount, FLOAT*& out);
void dihLocs(vector<Atom*> &atoms, FLOAT* &out, int s, int d);
void safeCombine(FLOAT **in, FLOAT* out, FLOAT *dim, int d, int stride);
void MatMult(FLOAT **a, FLOAT **b, FLOAT **c);
void closestToFirst(vector<Atom*> &atoms, int indOut[],FLOAT* N, bool *periodic, FLOAT* perDim);
void setMidp(FLOAT* out, FLOAT* in1, FLOAT* in2, bool *periodic, FLOAT* perDim);
#ifdef ipp
inline FLOAT** alloc2d(int length, int n) {
    FLOAT** out=(FLOAT**)malloc(sizeof(FLOAT*)*n);
    for (int i=0; i<n; ++i) {
        #ifdef ipp
                out[i]=IPPF1(ippsMalloc)(length);
        #else
                out[i]=(FLOAT*)malloc(sizeof(FLOAT)*length);
        #endif
    }
    return out;
}
inline void free2d(FLOAT** in, int n) {
    for (int i=0; i<n; ++i) {
        #ifdef ipp
                ippsFree(in[i]);
        #else
                free(in[i]);
        #endif
    }
    free(in);
}
#endif
/*out and normed out should have size 3*c*N, N is number of grid points,
 c is the number of centers, and thus the number of spirals to build  */

#define NULL 0

class elements
{
    public:
    elements();
    map<FLOAT,string> el; //Maps mass to element name
    map<FLOAT,string> m2; //Maps mass to element detailed name
};

#endif /*INC_H_*/
```

Additional supporting functions that are commonly used but need not be associated with a specific class type are included defined separately.

Math function definitions **vMath.cpp**

```
#include "inc.h"

void normalizeVecs(FLOAT* v, int c) {
    #ifdef ipp
        FLOAT* n=IPPF1(ippsMalloc)(c);
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        IPPF1(ippmL2Norm_va)(v,stride0,stride2,n,3,c);
        FLOAT* in=IPPF1(ippsMalloc)(c);
        IPPFIX(ippsInv)(n,in,c);
        ippsFree(n);
        FLOAT* tmp=IPPF1(ippsMalloc)(3*c);
        IPPF1(ippmMul_vaca)(v,stride0,stride2,in,stride2,tmp,stride0,stride2,3,c);
        ippsFree(in);
        for (int i=0; i<3*c; ++i)
                v[i]=tmp[i];
        ippsFree(tmp);
    #else
        FLOAT* n=(FLOAT*)malloc(c*sizeof(FLOAT));
        #pragma omp parallel
        {
                #pragma omp for
                for (int i=0; i<c; ++i) {
                        n[i]=0;
                        for (int j=0; j<3; ++j)
                                n[i]+=pow(v[3*i+j],2);
                }
                #pragma omp for
                for (int i=0; i<c; ++i) {
                        n[i]=sqrt(n[i]);
                        for (int j=0; j<3; ++j)
                                v[3*i+j]/=n[i];
                }
        }
        free(n);
    #endif
}

void relVecsF2R(vector<Atom*> atoms, FLOAT* out, int s, int d) {
    FLOAT* pos[atoms.size()];
    FLOAT* outPos[atoms.size()-1];
    for (int i=0; i<atoms.size(); ++i) {
        pos[i]=atoms[i]->getPos(s);
    }
    for (int i=0; i<atoms.size()-1; ++i) {
        outPos[i]=out+3*d*i;
    }
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        IPPF2(ippmSub_vav,L)(pos+1,0,stride2,pos[0],stride2,outPos,0,stride2,3*d,atoms.size()-1);
    #else
        #pragma omp parallel for
                for (int i=0; i<atoms.size()-1; ++i) {
                        for (int j=0; j<d; j++) {
                                for (int k=0; k<3; k++) {
                                        outPos[i][j*3+k]=pos[i+1][3*j+k]-pos[0][3*j+k];
                                }
                        }
                }
    #endif
    for (int i=0; i<3; ++i) {
        if (atoms[0]->parent->parent->periodic[i]) {
                #pragma omp parallel for
                for (int j=0; j<d*(atoms.size()-1); ++j) {
```

```
                        int relPd=atoms[0]->parent->parent->perDim[3*s+j/(atoms.size()-
    1)+i]/2.0;
                        if (fabs(double(out[j*3+i]))>relPd) {
                                if (out[j*3+i]>0)
                                        out[j*3+i]-=relPd;
                                else
                                        out[j*3+i]+=relPd;
                        }
                }
        }
    }
}

void relVecs(vector<Atom*> &atoms, FLOAT* out, int s, int d) {  //out should be preallocated with
    3*(c-1)*d elements
    FLOAT* pos[atoms.size()];
    FLOAT* outPos[atoms.size()-1];
    for (int i=0; i<atoms.size(); ++i) {
        pos[i]=atoms[i]->getPos(s);
    }
    for (int i=0; i<atoms.size()-1; ++i) {
        outPos[i]=out+3*d*i;
    }
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        IPPF2(ippmSub_vava,L)(pos+1,0,stride2,pos,0,stride2,outPos,0,stride2,3*d,atoms.size()-1);
    #else
        #pragma omp parallel for
        for (int i=0; i<atoms.size()-1; ++i) {
                for (int j=0; j<d; j++) {
                        for (int k=0; k<3; k++)
                                outPos[i][j*3+k]=pos[i+1][3*j+k]-pos[i][3*j+k];
                }
        }
    #endif
    Particle* P=atoms[0]->parent->parent;
    FLOAT relPd;
    for (int i=0; i<3; ++i) {
        if (P->periodic[i]) {
                #pragma omp parallel for private(relPd)
                for (int j=0; j<d*(atoms.size()-1); ++j) {
                        relPd=P->perDim[3*s+j/(atoms.size()-1)+i]/2.0;
                        if (fabs(double(out[j*3+i]))>relPd) {
                                if (out[j*3+i]>0)
                                        out[j*3+i]-=relPd;
                                else
                                        out[j*3+i]+=relPd;
                        }
                }
        }
    }
}

void dihLocs(vector<Atom*> &atoms, FLOAT* &out, int s, int d) {
    Particle* P=atoms[0]->parent->parent;
    int n=atoms.size();
    FLOAT** pos=(FLOAT**)malloc(sizeof(FLOAT*)*n);
    for (int i=0; i<n; ++i) {
        pos[i]=atoms[i]->getPos(s);
    }
    FLOAT t1,t2,t3,halfW;
    for (int i=0; i<n-3; ++i) {
        FLOAT* O=out+3*d*i;
        for (int j=0; j<3*d; ++j) {
                if (P->periodic[j%3]) {
```

```
                        halfW=P->perDim[s*3+(j/3)*3+j%3]/2.0;
                        t1=pos[i][j]-pos[i+1][j];
                        t2=pos[i+2][j]-pos[i+3][j];
                        if (fabs(t1)>halfW) {
                                t1=(pos[i][j]+pos[i+1][j])/2.0+halfW;
                        } else {
                                t1=(pos[i][j]+pos[i+1][j])/2.0;
                        }
                        if (fabs(t2)>halfW) {
                                t2=(pos[i+2][j]+pos[i+3][j])/2.0+halfW;
                        } else {
                                t2=(pos[i+2][j]+pos[i+3][j])/2.0;
                        }
                        t3=t1-t2;
                        if (fabs(t3)>halfW) {
                                O[j]=(t1+t2)/2.0+halfW;
                        } else {
                                O[j]=(t1+t2)/2.0;
                        }
                } else
                        O[j]=(pos[i][j]+pos[i+1][j]+pos[i+2][j]+pos[i+3][j])/4.0;
        }
    }
}

void radSpir(FLOAT* centers, FLOAT radius, int c, int N, FLOAT* points, FLOAT* iPoints) {
    FLOAT inc=PI*(3.0-sqrt(5.0));
    FLOAT off=2.0/FLOAT(N);
    #pragma omp parallel for
    for (int j=0; j<N; ++j) {
        iPoints[3*j+1]=FLOAT(j)*off-1+(off/2);
        FLOAT R=sqrt(1-pow(iPoints[3*j+1],2));
        FLOAT phi=FLOAT(j)*inc;
        iPoints[3*j]=cos(phi)*R;
        iPoints[3*j+2]=sin(phi)*R;
    }
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        FLOAT* rPoints=IPPF1(ippsMalloc)(3*N);
        IPPF1(ippsMulC)(iPoints,radius,rPoints,3*N);
        #pragma omp parallel for
        for (int j=0; j<c; ++j)

        IPPF1(ippmSub_vav)(rPoints,stride0,stride2,centers+3*j,stride2,points+3*j,c*stride0,strid
    e2,3,N);
        ippsFree(rPoints);
    #else
    #pragma omp parallel for
        for (int i=0; i<c; ++i) {
                for (int j=0; j<N; ++j) {
                        for (int k=0; k<3; ++k) {
                                points[c*3*j+3*i+k]=iPoints[3*j+k]*radius-centers[3*i+k];
                        }
                }
        }
    #endif
}

int rectGrid(FLOAT* dims, FLOAT radius, FLOAT* heights, int dCount, FLOAT*& out) {
    FLOAT y=dims[2],yhi=dims[3],x,xhi=dims[1];
    bool odd=0;
    int pointCount=0;
    //count points to allocate
    do {
```

```
        y+=(2*radius*sin(PI/4.0));
        if (odd) {
                x=dims[0]+2*radius*cos(PI/4.0);
                odd=0;
        } else {
                x=dims[0];
                odd=1;
        }
        do {
                ++pointCount;
                x+=(2*radius);
        } while (x<xhi);
    } while (y<yhi);
    #ifdef ipp
        out=IPPF1(ippsMalloc)(pointCount*dCount*3);
    #else
        out=(FLOAT*)malloc(pointCount*dCount*3*sizeof(FLOAT));
    #endif
    y=dims[2]; pointCount=0;
    //now create points
    do {
        y+=(2*radius*sin(PI/4.0));
        if (odd) {
                x=dims[0]+2*radius*cos(PI/4.0);
                odd=0;
        } else {
                x=dims[0];
                odd=1;
        }
        do {
                int pInd=dCount*pointCount*3;
                for (int i=0; i<dCount; ++i) {
                        int i3=3*i;
                        out[pInd+i3]=x;
                        out[pInd+i3+1]=y;
                        out[pInd+i3+2]=heights[i];
                }
                ++pointCount;
                x+=(2*radius);
        } while (x<xhi);
    } while (y<yhi);
    return pointCount;
}

void closestToFirst(vector<Atom*> &atoms, int indOut[], FLOAT* N, bool* periodic, FLOAT* perDim)
    {
    int stride2=sizeof(FLOAT);
    FLOAT** pos=(FLOAT**)malloc(sizeof(FLOAT*)*atoms.size());
    FLOAT** rel=alloc2d(3,atoms.size()-1);
    for (int i=0; i<atoms.size(); ++i)
        pos[i]=atoms[i]->getPos();
    IPPF2(ippmSub_vva,L)(pos[0],stride2,pos+1,0,stride2,rel,0,stride2,3,atoms.size()-1);
    for (int i=0; i<3; ++i) {
        if (periodic[i]) {
                for (int j=0; j<atoms.size()-1; ++j) {
                        if (fabs(rel[j][i])>perDim[i]) {
                                if (rel[j][i]<0)
                                        rel[j][i]+=perDim[i];
                                else
                                        rel[j][i]-=perDim[i];
                        }
                }
        }
    }
    IPPF2(ippmL2Norm_va,L)(rel,0,stride2,N,3,atoms.size()-1);
```

```
    IPPF2(ippsSortIndexAscend,I)(N,indOut,atoms.size()-1);
    free2d(rel,atoms.size()-1);
    free(pos);
}

void setMidp(FLOAT* out, FLOAT* in1, FLOAT* in2, bool* periodic, FLOAT* perDim) {
    out[0]=(in1[0]+in2[0])/2.0;
    out[1]=(in1[1]+in2[1])/2.0;
    out[2]=(in1[2]+in2[2])/2.0;
    FLOAT rel;
    for (int i=0; i<3; ++i) {
        if (periodic[i]) {
                rel=in1[i]-in2[i];
                if (fabs(rel)>perDim[i]/2.0) {
                        out[i]+=perDim[i]/2.0;
                }
        }
    }
}
```

B.2 Element Classes

Elements of a simulation include atoms, bonds, etc. The need to identify their context within list classes is common to all the elements, and these features are introduced via a template base class from which the subsequent classes are derived.

Element Base Class **Elem.h**

```
#ifndef ELEM_H_
#define ELEM_H_

template <typename E, typename L>
class Elem
{
public:
    Elem() { }
    virtual ~Elem() 66

    ElemList<E>* parent; //Link to the list containing the element

    E** lPointer;               //Pointer to this element in the parent list array
    int lIndex;                 //Index of this element in the parent list array

    void setInd(E** lp, int li); //Sets the indices for this element
    unsigned int getInd();                      //Returns lIndex
};

template <typename E, typename L>
void Elem<E,L>::setInd(E** lp, int li) {
    lPointer=lp;
    lIndex=li;
}

template <typename E, typename L>
unsigned int Elem<E,L>::getInd() {
    return lIndex;
}
```

```
#endif /*ELEM_H_*/
```

Atom class declaration **Atom.h**

```
#ifndef ATOM_H_
#define ATOM_H_

class Atom : protected Elem<Atom,AtomList>
{
public:
//Constructor/Destructor and base class calls
    Atom();
    Atom(FLOAT*, FLOAT, atomT*);
    Atom(FLOAT x, FLOAT y, FLOAT z, FLOAT ch, atomT* t);
    ~Atom();

    //Public functions from template class
    using Elem<Atom,AtomList>::setInd; //(Atom** lp, int li);
    using Elem<Atom,AtomList>::getInd;
    using Elem<Atom,AtomList>::lIndex;
    using Elem<Atom,AtomList>::lPointer;
    using Elem<Atom,AtomList>::parent;

    //Positional elements and accessors
    FLOAT* pos; //contains initial position from datafile, and subsequent snapshot positions
    void setPos(FLOAT*,int k=0); //Sets initial position by default, or position for simulation
    snapshot k
    void setPos(FLOAT*,int,int); //First int is the number of 3vectors, the second int is the
    dump count of the first one
    void getPos(FLOAT*,int k=0); //Gets initial position by default, or position for simulation
    snapshot k
    FLOAT* getPos(int k=0);              //Similar to previous, returns pointer instead of
    setting pointer
    FLOAT* operator[](int);              //Overloads [] operator to behavior of prior function
    unsigned int fileInd;                //
    map<unsigned int,Atom*> atmInd; //used for correlating indices when writing files, index
    points to atom

    bool isA(atomT* t) { if (t==type) return 1; else return 0; } //quick check for atom type

    //Properties
    FLOAT charge;
    atomT* type;
    Molecule* mol;

    //Dump data handling
    int dumps;          //# of simulation snapshots stored
    void resDumps(int);  //reserved memory for simulation snapshots

    //inputer outputer
    void writeData(ostream&,map<int,int>&, unsigned int oInd, int k=0); //writes atom position to
    data file
    void writeMol2(ostream&,map<int,int>&,elements& el, unsigned int oInd, int k=0); //writes
    atom position to mol2 file

    //Bond data
    list<Bond*> bonds;
    void addBond(Bond*);
    bool remBond(Bond*);

    //Angle data
    list<Angle*> angles;
```

```
    void addAngle(Angle*);
    bool remAngle(Angle*);

    //Dih Data
    list<Dihed*> dihs;
    void addDih(Dihed*);
    bool remDih(Dihed*);

    //Imp Data
    list<Improper*> imps;
    void addImp(Improper*);
    bool remImp(Improper*);

    //Returns connectivity data, bonds, angles, etc...
    void collectConn(set<unsigned int> &b, set<unsigned int> &a, set<unsigned int> &d,
    set<unsigned int> &i);
    void collectAng(set<unsigned int> &a);

    //Removes all connections to this atom
    void clearConn();

    bool getNet(set<int> &net, Atom* call);

    //Utilities
    int getBonded(vector<Atom*>&); //Gets a list of atoms bonded to this one, returns count
};
#endif /*ATOM_H_*/
```

Atom function definitions **Atom.cpp**

```
include "inc.h"

Atom::Atom()
{
    #ifdef ipp
        pos=IPPF1(ippsMalloc)(3);
    #else
        pos=(FLOAT*)malloc(3*sizeof(FLOAT));
    #endif
    dumps=0;
    mol=NULL;
}

Atom::~Atom()
{
    while (bonds.size()>0)
        parent->parent->bonds.remElem((*bonds.begin())->getInd());
    while (angles.size()>0)
        parent->parent->angles.remElem((*angles.begin())->getInd());
    while (dihs.size()>0)
        parent->parent->dihedrals.remElem((*dihs.begin())->getInd());
    while (imps.size()>0)
        parent->parent->impropers.remElem((*imps.begin())->getInd());
    if (mol!=NULL)
        mol->remAtom(this);
    #ifdef ipp
        ippsFree(pos);
    #else
        free(pos);
    #endif
}

Atom::Atom(FLOAT p[], FLOAT ch, atomT* t) {
    #ifdef ipp
```

```
        pos=IPPF1(ippsMalloc)(3);
        IPPF1(ippsCopy)(p,pos,3);
    #else
        pos=(FLOAT*)malloc(3*sizeof(FLOAT));
        for (int i=0; i<3; i++)
                pos[i]=p[i];
    #endif
    type=t;
    charge=ch;
    dumps=0;
    mol=NULL;
}

Atom::Atom(FLOAT x, FLOAT y, FLOAT z, FLOAT ch, atomT* t) {
    #ifdef ipp
        pos=IPPF1(ippsMalloc)(3);
    #else
        pos=(FLOAT*)malloc(3*sizeof(FLOAT));
    #endif
    pos[0]=x;
    pos[1]=y;
    pos[2]=z;
    type=t;
    charge=ch;
    dumps=0;
    mol=NULL;
}

void Atom::resDumps(int k) {
    #ifdef ipp
        FLOAT* nPos=IPPF1(ippsMalloc)(3*(dumps+k+1));
        IPPF1(ippsZero)(nPos,3*dumps+k+1);
        IPPF1(ippsCopy)(pos,nPos,3*(dumps+1));
        ippsFree(pos);
        pos=nPos;
    #else
        pos=(FLOAT*)realloc(pos,(dumps+k+1)*3*sizeof(FLOAT));
    #endif
    dumps+=k;
}

void Atom::setPos(FLOAT p[], int k) {
    if (k<0) k=dumps+1+k;
    if (k>dumps) resDumps(k-dumps);
    #ifdef ipp
        IPPF1(ippsCopy)(p,pos+3*k,3);
    #else
        for (int i=0; i<3; ++i)
                pos[3*k+i]=p[i];
    #endif
}

void Atom::setPos(FLOAT p[], int c, int k) {
    if ((k+c)>dumps) resDumps((k+c)-dumps);
    #ifdef ipp
        IPPF1(ippsCopy)(p,pos+3*k,3*c);
    #else
        for (int i=k; i<k+c; ++i) {
                for (int j=0; j<3; ++j)
                        pos[3*i+j]=p[3*(k-i)+j];
        }
    #endif
}

void Atom::getPos(FLOAT p[], int k) {
```

```
    if (k<0)
        k=dumps+1+k;
    if (k>dumps) {
        cerr << "Trying to get a dump location that doesn't exist";
        abort();
    }
    #ifdef ipp
        IPPF1(ippsCopy)(pos+3*k,p,3);
    #else
        for (int i=0; i<3; i++)
                p[i]=pos[3*k+1];
    #endif
}

FLOAT* Atom::getPos(int k) {
    if (k<0)
        k=dumps+1+k;
    return pos+3*k;
}

FLOAT* Atom::operator[](int k) {
    return getPos(k);
}

void Atom::writeData(ostream& out, map<int,int>& molInd, unsigned int oInd, int k) {
    fileInd=oInd;
    if (parent->parent->molecular) {
        if (molInd.find(mol->lIndex)==molInd.end()) {
                if (molInd.empty())
                        molInd[mol->lIndex]=1;
                else {
                        int max=0;
                        for (map<int,int>::iterator i=molInd.begin(); i!=molInd.end(); i++) {
                                if (i->second>max) {
                                        max=i->second;
                                }
                        }
                        molInd[mol->lIndex]=max+1;
                }
        }
        out << setw(15) << left << molInd[mol->lIndex];
    }
    out << setw(4) << left << parent->parent->aTypes[type];
    out << setw(10) << setprecision(4) << left << charge;
    FLOAT* opos=getPos(k);
    for (int i=0; i<3; ++i)
        out << setw(20) << setprecision(12) << left << opos[i];
    for (int i=0; i<3; ++i) {
        if (!(parent->parent->periodic[i])) {
                out << " 0";
                continue;
        }
        int flag=-1;
        Atom *l, *r;
        for (std::list<Bond*>::iterator j=bonds.begin(); j!=bonds.end(); ++j) {
                if ((*j)->isBnd(i,l,r)) {
                        if (this==l)
                                flag=0;
                        else
                                flag=1;
                        break;
                }
        }
        if (flag!=-1) {
                out << " " << flag;
```

```
                continue;
        }
        if (flag==-1 || flag==0) {
                out << " 0";
        } else {
                out << " 1";
        }
    }
    out << endl;
}

void Atom::writeMol2(ostream& out, map<int,int>& molInd, elements& el, unsigned int oInd, int k)
    {
    fileInd=oInd;
    if (molInd.find(mol->lIndex)==molInd.end()) {
        int max=0;
        for (map<int,int>::iterator i=molInd.begin(); i!=molInd.end(); i++) {
                if (i->second>max) {
                        max=i->second;
                }
        }
        molInd[mol->lIndex]=max+1;
    }
    out << setw(2) << setfill (' ') << left << el.el[type->mass] << ' ';
    FLOAT* oPos=getPos(k);
    for (int i=0; i<3; ++i)
        out << setw(15) << setprecision(6) << left << oPos[i];
    out << setw(5) << left << el.m2[type->mass] << molInd[mol->lIndex] << " LIG" << setw(5) <<
    setfill (' ') << left << molInd[mol->lIndex] << " 0.000" << endl;
}

void Atom::addBond(Bond* b) {
    bonds.push_back(b);
}

bool Atom::remBond(Bond* b) {
    for (list<Bond*>::iterator i=bonds.begin(); i!=bonds.end(); ++i) {
        if ((*i)==b) {
                bonds.erase(i);
                return 1;
        }
    }
    DEB(Could not find bond to erase)
    return 0;
}

void Atom::addAngle(Angle* a) {
    angles.push_back(a);
}

bool Atom::remAngle(Angle* a) {
    for (list<Angle*>::iterator i=angles.begin(); i!=angles.end(); ++i) {
        if (*i==a) {
                angles.erase(i);
                return 1;
        }
    }
    DEB(Could not find angle to erase)
    return 0;
}

void Atom::addDih(Dihed* d) {
    dihs.push_back(d);
}
```

```
bool Atom::remDih(Dihed* d) {
    for (list<Dihed*>::iterator i=dihs.begin(); i!=dihs.end(); ++i) {
        if (*i==d) {
                dihs.erase(i);
                return 1;
        }
    }
    return 0;
}

void Atom::addImp(Improper* i) {
    imps.push_back(i);
}

bool Atom::remImp(Improper* im) {
    for (list<Improper*>::iterator i=imps.begin(); i!=imps.end(); ++i) {
        if (*i==im) {
                imps.erase(i);
                return 1;
        }
    }
    return 0;
}

int Atom::getBonded(vector<Atom*>& out) {
    for (list<Bond*>::iterator i=bonds.begin(); i!=bonds.end(); ++i) {
        out.push_back((*i)->getOther(this));
    }
    return bonds.size();
}

void Atom::collectConn(set<unsigned int> &b, set<unsigned int> &a, set<unsigned int> &d,
    set<unsigned int> &i) {
    for (list<Bond*>::iterator j=bonds.begin(); j!=bonds.end(); ++j)
        b.insert((*j)->lIndex);
    for (list<Angle*>::iterator j=angles.begin(); j!=angles.end(); ++j)
        a.insert((*j)->lIndex);
    for (list<Dihed*>::iterator j=dihs.begin(); j!=dihs.end(); ++j)
        d.insert((*j)->lIndex);
    for (list<Improper*>::iterator j=imps.begin(); j!=imps.end(); ++j)
        i.insert((*j)->lIndex);
}

void Atom::collectAng(set<unsigned int> &a) {
    for (list<Angle*>::iterator j=angles.begin(); j!=angles.end(); ++j)
        a.insert((*j)->lIndex);
}


void Atom::clearConn() {
    bonds.clear();
    angles.clear();
    dihs.clear();
    imps.clear();
}
```

Bond class declaration **Bond.h**

```
#ifndef BOND_H_
#define BOND_H_

class BondList;

class Bond : protected Elem<Bond,BondList>
```

```
{
public:
    //construction/destruction
    Bond();
    Bond(Atom*, Atom*, bondT*);
    virtual ~Bond();

    //Bond properties
    Atom* atoms[2]; //pointers to the two atoms of the bond
    bondT* type;//bond type

    //Inherited functions made public
    using Elem<Bond,BondList>::setInd;
    using Elem<Bond,BondList>::getInd;
    using Elem<Bond,BondList>::parent;
    using Elem<Bond,BondList>::lIndex;

    void calcDist(int, FLOAT*, int d=0);  /* First parameter indicates start of calculation,
    0=datafile, 1=firstdump
    second parameter is output of distances, 3rd parameter dictates how many snapshots to go
    through*/
    Atom* getOther(Atom*);      //returns pointer to the other atom of the bond
    bool isBnd(int dim, Atom*& l, Atom*& r, int k=0); //determines if bond spans periodic cell
    edge

    void calcEnergy(FLOAT* out, int s=0, int d=1); //calculates bond energy

    void writeBondData(ostream&);
    void writeBondMol2(ostream&);

    bool verify(); //quick check to make sure bond doesn't span two different molecules
    //sometimes it's okay if it does

    void rereg(); //Reregisters the bond with the substituent atoms
};

Angle* operator+(Bond &b1, Bond &b2); //Combines two bonds into an angle
#endif /*BOND_H_*/
```

Bond Class Function Definitions **Bond.cpp**

```
include "inc.h"

Bond::Bond()
{
}

Bond::~Bond()
{
    if (!atoms[0]->remBond(this))
        cout << "Atom 0 didn't have bond" << endl;
    if (!atoms[1]->remBond(this))
        cout << "Atom 1 didn't have bond" << endl;
}

Bond::Bond(Atom* a1, Atom* a2, bondT* t) {
    atoms[0]=a1;
    atoms[0]->addBond(this);
    atoms[1]=a2;
    atoms[1]->addBond(this);
    type=t;
}

void Bond::calcDist(int s, FLOAT* out, int d) {
```

```
    FLOAT* loc1, *loc2;
    loc1=atoms[0]->getPos(s);
    loc2=atoms[1]->getPos(s);
    if ((s+d)>atoms[0]->dumps || (s+d)>atoms[1]->dumps) {
        out=NULL;
        return;
    }
    #ifdef ipp
        FLOAT* subs=IPPF1(ippsMalloc)((d+1)*3);
        IPPF1(ippsSub)(loc1,loc2,subs,(d+1)*3);
        parent->parent->periodicCheck(subs,s,d,1);
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        IPPF1(ippmL2Norm_va)(subs,stride0,stride2,out,3,d+1);
    #else
        FLOAT subs[(d+1)*3];
        for (int i=0; i<(d+1)*3; i++)
                subs[i]=loc2[i]-loc1[i];
        for (int i=0; i<d+1; i++)
                out[i]=sqrt(pow(subs[3*i],2)+pow(subs[3*i+1],2)+pow(subs[3*i+2],2));
    #endif
}

void Bond::writeBondData(ostream& out) {
    out << setw(4) << left << parent->parent->bTypes[type];
    out << setw(15) << left << atoms[0]->fileInd;
    out << setw(15) << left << atoms[1]->fileInd;
    out << endl;
}

void Bond::writeBondMol2(ostream& out) {
    out << setw(15) << left << atoms[0]->fileInd;
    out << setw(15) << left << atoms[1]->fileInd;
    out << 1 << endl;
}

Atom* Bond::getOther(Atom* t) {
    if (atoms[0]==t)
        return atoms[1];
    else
        return atoms[0];
}

bool Bond::isBnd(int dim, Atom*& l, Atom*& r, int k) {
    //Default case dim is 3, tells it to evaluate over all dimensions
    if (dim==3) {
        for (int i=0; i<3; ++i) {
                if (parent->parent->periodic[i]) {
                        if (isBnd(i,l,r,k))
                                return 1;
                }
        }
        return 0;
    }
    FLOAT dist=(atoms[1]->getPos(k))[dim]-(atoms[0]->getPos(k))[dim];
    if (fabs(dist)>parent->parent->perDim[3*k+dim]/2.0) {
        if (dist>0) {
                l=atoms[0];
                r=atoms[1];
        } else if (dist<0) {
                l=atoms[1];
                r=atoms[0];
        }
        return 1;
    } else
```

```
        return 0;
}

bool Bond::verify() {
    if (atoms[0]->mol!=atoms[1]->mol)
        return 0;
    else
        return 1;
}

void Bond::rereg() {
    atoms[0]->addBond(this);
    atoms[1]->addBond(this);
}

Angle* operator+(Bond &b1, Bond &b2) {
    Angle* out=NULL;
    if (b1.atoms[0] == b2.atoms[0]) {
        out = new Angle(b1.atoms[1],b1.atoms[0],b2.atoms[1]);
    } else if (b1.atoms[1] == b2.atoms[0]) {
        out = new Angle(b1.atoms[0],b1.atoms[1],b2.atoms[1]);
    } else if (b1.atoms[1] == b2.atoms[1]) {
        out = new Angle(b1.atoms[0],b1.atoms[1],b2.atoms[0]);
    } else if (b1.atoms[0] == b2.atoms[1]) {
        out = new Angle(b1.atoms[1],b1.atoms[0],b2.atoms[0]);
    }
    return out;
}
```

Angle class declaration **Angle.h**

```
#ifndef ANGLE_H_
#define ANGLE_H_

#include "Dihed.h"  //Required for operator+ casting

class AngleList;

class Angle : protected Elem<Angle,AngleList>
{
public:
//Constructors & Destructors
    Angle();
    Angle(Atom*, Atom*, Atom* , angleT*);
    Angle(Atom*, Atom*, Atom*);
    virtual ~Angle();

    //Pulls from base class
    using Elem<Angle,AngleList>::setInd;
    using Elem<Angle,AngleList>::getInd;
    using Elem<Angle,AngleList>::parent;
    using Elem<Angle,AngleList>::lIndex;

    Atom* atoms[3]; //atoms in the angle
    angleT* type;   //Angle type

    void writeAngleData(ostream&); //writes angle data, the type, and the three atoms involved

    bool isBnd(int dim, set<short>& left, set<short>&right); //Determines if angle is on
    boundary, if true, left provides atoms at left boundary, right provides atoms at right
    boundary
    bool verify(int k); //verifies angle connectivity by proximity checking

    void calcEnergy(FLOAT* out, int s, int d); //calculates potential energy of angle
```

```
    friend Dihed* operator+(Angle &ang1, Angle &ang2); //assembles dihedral from incoming angles

    void rereg();
};

#endif /*ANGLE_H_*/
```

Angle function definitions — Angle.cpp

```
#include "inc.h"

Angle::Angle()
{
}

Angle::~Angle()
{
    for (int i=0; i<3; ++i)
        atoms[i]->remAngle(this);
}

Angle::Angle(Atom* a1, Atom* a2, Atom* a3, angleT* t) {
    atoms[0]=a1;
    atoms[1]=a2;
    atoms[2]=a3;
    atoms[0]->addAngle(this);
    atoms[1]->addAngle(this);
    atoms[2]->addAngle(this);
    type=t;
}

Angle::Angle(Atom* a1, Atom* a2, Atom* a3) {
    atoms[0]=a1;
    atoms[1]=a2;
    atoms[2]=a3;
    atoms[0]->addAngle(this);
    atoms[1]->addAngle(this);
    atoms[2]->addAngle(this);
}

void Angle::writeAngleData(ostream& out) {
    out << setw(4) << left << parent->parent->angTypes[type];
    for (int i=0; i<3; ++i)
        out << setw(15) << left << atoms[i]->fileInd;
    out << endl;
}

bool Angle::isBnd(int dim, set<short>& left, set<short>& right) {
    FLOAT rel1=(atoms[1]->getPos())[dim]-(atoms[0]->getPos())[dim];
    FLOAT rel2=(atoms[1]->getPos())[dim]-(atoms[2]->getPos())[dim];
    FLOAT check=parent->parent->perDim[dim]/2.0;
    if (fabs(rel1)>check || fabs(rel2)>check) {
        FLOAT l=parent->parent->dim[2*dim];
        for (int i=0; i<3; ++i) {
                if ((atoms[i]->getPos())[dim]>(l+check))
                        right.insert(i);
                else
                        left.insert(i);
        }
        return 1;
    }
    return 0;
}
```

```
bool Angle::verify(int k) {
    FLOAT* pos[3];
    for (int i=0; i<3; ++i) {
        pos[i]=atoms[i]->getPos(k);
    }
    FLOAT rel[2][3];
    for (int i=0; i<2; ++i) {
        for (int j=0; j<3; ++j) {
                rel[i][j]=pos[i+1][j]-pos[i][j];
        }
    }
    for (int i=0; i<3; ++i) {
        if (parent->parent->periodic[i]) {
                for (int j=0; j<2; ++j) {
                        if (fabs(rel[j][i])>parent->parent->perDim[i]/2.0) {
                                if (rel[j][i]>0) {
                                        rel[j][i]-=parent->parent->perDim[i];
                                } else {
                                        rel[j][i]+=parent->parent->perDim[i];
                                }
                        }
                }
        }
    }
    for (int i=0; i<2; ++i) {
        FLOAT n=sqrt(pow(rel[i][0],2)+pow(rel[i][1],2)+pow(rel[i][2],2));
        if (n>4.0)
                return 0;
    }
    return 1;
}

void Angle::rereg() {
    for (int i=0; i<3; ++i) {
        atoms[i]->addAngle(this);
    }
}

Dihed* operator+(Angle &ang1, Angle &ang2) {
    Dihed* out=NULL;
    if (ang1.atoms[1]==ang2.atoms[0] && ang1.atoms[2]==ang2.atoms[1]) {
        out=new Dihed(ang1.atoms[0],ang1.atoms[1],ang1.atoms[2],ang2.atoms[2]);
    } else if (ang1.atoms[1]==ang2.atoms[2] && ang1.atoms[2]==ang2.atoms[1]) {
        out=new Dihed(ang2.atoms[0],ang2.atoms[1],ang2.atoms[2],ang1.atoms[0]);
    } else if (ang1.atoms[0]==ang2.atoms[1] && ang1.atoms[1]==ang2.atoms[0]) {
        out=new Dihed(ang1.atoms[2],ang2.atoms[0],ang2.atoms[1],ang2.atoms[2]);
    } else if (ang1.atoms[0]==ang2.atoms[1] && ang1.atoms[1]==ang2.atoms[2]) {
        out=new Dihed(ang2.atoms[0],ang1.atoms[0],ang1.atoms[1],ang1.atoms[2]);
    }
    return out;
}
```

Dihedral class declaration **Dihed.h**

```
#ifndef DIHED_H_
#define DIHED_H_

class DihedList;

class Dihed : protected Elem<Dihed,DihedList>
{
public:
//Constructors & Destructors
    Dihed();
```

```
    Dihed(Atom*, Atom*, Atom*, Atom*, dihedT*);
    Dihed(Atom*, Atom*, Atom*, Atom*);
    virtual ~Dihed();

    //Member elements
    Atom* atoms[4];
    dihedT* type;

    //Derived elements
    using Elem<Dihed,DihedList>::setInd;
    using Elem<Dihed,DihedList>::getInd;
    using Elem<Dihed,DihedList>::parent;
    using Elem<Dihed,DihedList>::lIndex;

    void writeDihData(ostream&); //writes type and atom numbers to file
    bool leftRight(int dim, set<short>& left, set<short>& right); //determines if crossing a
    boundary, if so, puts left boundary atoms in left, right boundary atoms in right
    bool verify(int k); //verifies that dihedral atoms are appropriate by proximity for timestep
    k (0=data)

    void calcEnergy(FLOAT* out, int s=0, int d=1); //calculates energy of dihedral

    void rereg();
};

#endif /*DIHED_H_*/
```

Dihedral function definitions **Dihed.cpp**

```
#include "inc.h"

Dihed::Dihed()
{
}

Dihed::~Dihed()
{
    for (int i=0; i<4; ++i)
        atoms[i]->remDih(this);
}

Dihed::Dihed(Atom* a1, Atom* a2, Atom* a3, Atom* a4, dihedT* t) {
    atoms[0]=a1;
    atoms[1]=a2;
    atoms[2]=a3;
    atoms[3]=a4;
    type=t;
    for (int i=0; i<4; ++i)
        atoms[i]->addDih(this);
}

Dihed::Dihed(Atom* a1, Atom* a2, Atom* a3, Atom* a4) {
    atoms[0]=a1;
    atoms[1]=a2;
    atoms[2]=a3;
    atoms[3]=a4;
    for (int i=0; i<4; ++i)
        atoms[i]->addDih(this);
}

void Dihed::writeDihData(ostream& out) {
    out << setw(4) << left << parent->parent->dTypes[type];
    for (int i=0; i<4; ++i)
        out << setw(15) << left << atoms[i]->fileInd;
```

```
    out << endl;
}

bool Dihed::leftRight(int dim, set<short>& left, set<short>& right) {
    FLOAT rel[3];
    rel[0]=(atoms[1]->getPos())[dim]-(atoms[0]->getPos())[dim];
    rel[1]=(atoms[2]->getPos())[dim]-(atoms[1]->getPos())[dim];
    rel[2]=(atoms[3]->getPos())[dim]-(atoms[2]->getPos())[dim];
    FLOAT check=parent->parent->perDim[dim]/2.0;
    if (fabs(rel[0])>check || fabs(rel[1])>check || fabs(rel[2])>check) {
        FLOAT l=parent->parent->dim[2*dim];
        for (int i=0; i<4; ++i) {
                if ((atoms[i]->getPos())[dim]>(l+check))
                        right.insert(i);
                else
                        left.insert(i);
        }
        return 1;
    }
    return 0;
}

bool Dihed::verify(int k) {
    FLOAT* pos[4];
    for (int i=0; i<4; ++i) {
        pos[i]=atoms[i]->getPos(k);
    }
    FLOAT rel[3][3];
    for (int i=0; i<3; ++i) {
        for (int j=0; j<3; ++j) {
                rel[i][j]=pos[i+1][j]-pos[i][j];
        }
    }
    for (int i=0; i<3; ++i) {
        if (parent->parent->periodic[i]) {
                for (int j=0; j<3; ++j) {
                        if (fabs(rel[j][i])>parent->parent->perDim[i]/2.0) {
                                if (rel[j][i]>0) {
                                        rel[j][i]-=parent->parent->perDim[i];
                                } else {
                                        rel[j][i]+=parent->parent->perDim[i];
                                }
                        }
                }
        }
    }
    for (int i=0; i<3; ++i) {
        FLOAT n=sqrt(pow(rel[i][0],2)+pow(rel[i][1],2)+pow(rel[i][2],2));
        if (n>4.0)
                return 0;
    }
    return 1;
}

void Dihed::rereg() {
    for (int i=0; i<4; ++i) {
        atoms[i]->addDih(this);
    }
}
```

Improper class declaration **Improper.h**

```
#ifndef IMPROPER_H_
#define IMPROPER_H_
```

```
class Improper : protected Elem<Improper,ImpList>
{
public:
    //Constructors/Destructors
    Improper();
    Improper(Atom*,Atom*,Atom*,Atom*,impT*);
    virtual ~Improper();

    //Properties
    Atom* atoms[4];
    impT* type;

    //Public derived members
    using Elem<Improper,ImpList>::setInd;
    using Elem<Improper,ImpList>::getInd;
    using Elem<Improper,ImpList>::parent;
    using Elem<Improper,ImpList>::lIndex;

    void writeImpData(ostream&);

    bool leftRight(int dim, set<short>& l, set<short>& r); //Determine which atoms are over the
    periodic boundary
    bool verify(int k);

    void calcEnergy(FLOAT* out, int s=0, int d=1);

    void rereg();
};

#endif /*IMPROPER_H_*/
```

Improper Function Definitions **Improper.cpp**

```
#include "inc.h"

Improper::Improper()
{
}

Improper::~Improper()
{
    for (int i=0; i<4; i++)
        atoms[i]->remImp(this);
}

Improper::Improper(Atom* a1, Atom* a2, Atom* a3, Atom* a4, impT* t) {
    atoms[0]=a1;
    atoms[1]=a2;
    atoms[2]=a3;
    atoms[3]=a4;
    type=t;
    for (int i=0; i<4; ++i)
        atoms[i]->addImp(this);
}

void Improper::writeImpData(ostream& out) {
    out << setw(4) << left << parent->parent->iTypes[type];
    for (int i=0; i<4; ++i)
        out << setw(15) << left << atoms[i]->fileInd;
    out << endl;
}

bool Improper::leftRight(int dim, set<short>& left, set<short>& right) {
```

```
    FLOAT rel[3];
    rel[0]=(atoms[1]->getPos())[dim]-(atoms[0]->getPos())[dim];
    rel[1]=(atoms[2]->getPos())[dim]-(atoms[1]->getPos())[dim];
    rel[2]=(atoms[3]->getPos())[dim]-(atoms[2]->getPos())[dim];
    FLOAT check=parent->parent->perDim[dim]/2.0;
    if (fabs(rel[0])>check || fabs(rel[1])>check || fabs(rel[2])>check) {
        FLOAT l=parent->parent->dim[2*dim];
        for (int i=0; i<4; ++i) {
                if ((atoms[i]->getPos())[dim]>(l+check))
                        right.insert(i);
                else
                        left.insert(i);
        }
        return 1;
    }
    return 0;
}

bool Improper::verify(int k) {
    FLOAT* pos[4];
    for (int i=0; i<4; ++i) {
        pos[i]=atoms[i]->getPos(k);
    }
    FLOAT rel[3][3];
    for (int i=0; i<3; ++i) {
        for (int j=0; j<3; ++j) {
                rel[i][j]=pos[i+1][j]-pos[i][j];
        }
    }
    for (int i=0; i<3; ++i) {
        if (parent->parent->periodic[i]) {
                for (int j=0; j<3; ++j) {
                        if (fabs(rel[j][i])>parent->parent->perDim[i]/2.0) {
                                if (rel[j][i]>0) {
                                        rel[j][i]-=parent->parent->perDim[i];
                                } else {
                                        rel[j][i]+=parent->parent->perDim[i];
                                }
                        }
                }
        }
    }
    for (int i=0; i<3; ++i) {
        FLOAT n=sqrt(pow(rel[i][0],2)+pow(rel[i][1],2)+pow(rel[i][2],2));
        if (n>4.0)
                return 0;
    }
    return 1;
}



void Improper::rereg() {
    for (int i=0; i<4; ++i)
        atoms[i]->addImp(this);
}
```

The ‘molecule’ class is not derived off of the ‘Elem’ template because it has many special functions that aren’t amenable. A “molecule” in this context and in LAMMPS simply represents a collection of atoms, and the subdivision of atomic groups is generally

used for convenience but has no impact on the structure or dynamics.

Molecule class declaration — **Molecule.h**

```
#ifndef MOLECULE_H_
#define MOLECULE_H_

class MolecList;

class Molecule
{
public:
    //constructors/destructors
    Molecule();
    Molecule(MolecList*);
    virtual ~Molecule();

    list<Atom*> atoms;  //Atoms contained in this molecule
    ElemList<Molecule>* parent;  //Particle container pointer
    Molecule** lPointer; //Pointer to this guys position in the list of molecules
    int lIndex;                 //This guys index in the list of molecules

    void setInd(Molecule**, int);
    void addAtom(Atom*);
    void addAtomRec(Atom*);      //recursively adds atoms bonded to supplied atom
    bool safeAdd(Atom*); //return 0 if atom already in molecule, otherwise returns 1
    bool remAtom(Atom*);
    void remAtom(list<Atom*>::iterator);

    int calcDihedChain(FLOAT*&,int s=0, int d=0);//Used to return positions of carbon atoms in
    the chain, for bonded alkylsilane
    int calcChainDir(FLOAT*&, FLOAT* centers,int s=0, int d=0);  //Determines mean direction of
    alkylsilane chain relative to supplied centers
    int calcHexDih(FLOAT*& dOut, FLOAT*& pOut , int s=0, int d=0, bool round=1);  //Determines
    dihedrals of a hexane solvent molecule
    bool leftRight(int dim, set<Atom*>& left, set<Atom*>& right); //Determines which atoms are
    over a periodic boundary

    void replicate(int dim, int n, Molecule* m[]);              //Replicates the molecule when
    replicating over periodic boundaries
    void transfer(Molecule* inp); //Transfer all atoms from here to inp

    bool verify();

    void bound(FLOAT ang, FLOAT space);
    void idHexChain(vector<Atom*> &out); //Generated list of carbon backbond atoms in hexane
    molecule

    void clearAtoms();                  //Removes atoms contained in the molecule

    void translate(FLOAT* del);         //Moves molecules by vector del
    void rotate(FLOAT* rot); //Rotates the atoms of the molecule using rotation matrix rot
    void findSilaneEnd(Atom* &silane, Atom* &firstC); //Gets point to the carbon atom bound to
    the silica surface
};

#endif /*MOLECULE_H_*/
```

Molecule function definitions — **Molecule.cpp**

```
#ifndef ipp
```

```
    #include "bMathLibs.h"
#endif
#include <algorithm>
#include "inc.h"

Molecule::Molecule()
{
}

Molecule::~Molecule()
{
    if (!atoms.empty()) {
        for(list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i)
                (*i)->mol=NULL;
    }
}

void Molecule::setInd(Molecule** lp, int li) {
    lPointer=lp;
    lIndex=li;
}

void Molecule::addAtom(Atom* a) {
    if (a->mol!=NULL)
        a->mol->remAtom(a);
    atoms.push_back(a);
    a->mol=this;
}

bool Molecule::safeAdd(Atom* a) {
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)==a)
                return 0;
    }
    atoms.push_back(a);
    a->mol=this;
    return 1;
}

void Molecule::addAtomRec(Atom* a) {
    if (safeAdd(a)) {
        vector<Atom*> bonded;
        a->getBonded(bonded);
        for (int i=0; i<bonded.size(); ++i) {
                addAtomRec(bonded[i]);
        }
    }
}

bool Molecule::remAtom(Atom* a) {
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if (*i==a) {
                remAtom(i);
                return 1;
        }
    }
    return 0;
}

void Molecule::remAtom(list<Atom*>::iterator e) {
    (*e)->mol=NULL;
    atoms.erase(e);
}

int Molecule::calcHexDih(FLOAT* &dOut, FLOAT* &pOut, int s, int d, bool round) {
```

```
    //TODO
    vector<Atom*> chain;
    idHexChain(chain);
    int l=chain.size();
    #ifdef ipp
        dOut=IPPF1(ippsMalloc)((l-3)*d);  // = ippsMalloc_64f(((l-3*d))
        pOut=IPPF1(ippsMalloc)((l-3)*d*3);
        FLOAT* dVec=IPPF1(ippsMalloc)((l-1)*3*d);
    #else
        dOut=(FLOAT*)malloc(sizeof(FLOAT)*(l-3)*d);
        pOut=IPPF1(ippsMalloc)((l-3)*d*3*sizeof(FLOAT));
        FLOAT* dVec=(FLOAT*)malloc(sizeof(FLOAT)*(l-1)*3*d);
    #endif
    relVecs(chain,dVec,s,d);
    normalizeVecs(dVec,(l-1)*d);
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        FLOAT* cVec=IPPF1(ippsMalloc)((l-2)*3*d);
        FLOAT* dots=IPPF1(ippsMalloc)((l-3)*d);
        for (int i=0; i<l-2; ++i)

        IPPF1(ippmCrossProduct_vava)(dVec+3*d*(i+1),stride0,stride2,dVec+3*d*i,stride0,stride2,cV
    ec+3*d*i,stride0,stride2,d);
        FLOAT* crN=IPPF1(ippsMalloc)((l-2)*d);
        IPPF1(ippmL2Norm_va)(cVec,stride0,stride2,crN,3,(l-2)*d);
        FLOAT* CR;
        for (int i=0; i<(l-2)*d; ++i) {
                CR=cVec+3*i;
                for (int j=0; j<3; ++j) {
                        CR[j]=CR[j]/crN[i];
                }
        }
        for (int i=0; i<l-3; ++i)

        IPPF1(ippmDotProduct_vava)(cVec+3*d*(i+1),stride0,stride2,cVec+3*d*i,stride0,stride2,dots
    +d*i,3,d);
        IPPFIX(ippsAcos)(dots,dOut,d*(l-3));
        IPPF2(ippsMulC,I)(R2D,dOut,d*(l-3));
        ippsFree(cVec);
        ippsFree(dVec);
        ippsFree(dots);
    #else
        for (int i=0; i<l-1; ++i) {
                for (int j=0; j<d; ++j) {
                        dVec[3*d*i+3*j+0] = pos[i+1][3*j+0] - pos[i][3*j+0];
                        dVec[3*d*i+3*j+1] = pos[i+1][3*j+1] - pos[i][3*j+1];
                        dVec[3*d*i+3*j+2] = pos[i+1][3*j+2] - pos[i][3*j+2];
                }
        }
    #endif
    dihLocs(chain,pOut,s,d);
    return l-3;
}

void Molecule::idHexChain(vector<Atom*> &out) {
    //TODO
    //Start by finding a terminal carbon
    int hydrogens;
    vector<Atom*> bonds;
    int b;
    atomT* H=parent->parent->aTypes[21];
    atomT* C=parent->parent->aTypes[20];
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)->isA(H))  //i is a hydrogen don't care
```

```
                continue;
        b=(*i)->getBonded(bonds); //get bonds, looking for 3 hydrogens
        hydrogens=0;
        for (int j=0; j<b; ++j) {
                if (bonds[j]->isA(H))
                        hydrogens++;
        }
        if (hydrogens==3) {   //case has 3 hydrogens, store it and leave
                out.push_back(*i);
                hydrogens=0;
                break;
        } else {  //yawn and move on
                bonds.clear();
        }
    }
    Atom* prev=NULL;
    while (hydrogens!=3) {
        if (out.size()>1) {
                prev=*(out.rbegin()+1); //Second to last atom in chain is previous
        }
        for (int i=0; i<b; ++i) {
                if (bonds[i]->isA(C) && bonds[i]!=prev) {
                        out.push_back(bonds[i]); //store it
                        bonds.clear();
                        b=(*out.rbegin())->getBonded(bonds); //get its bonds
                        hydrogens=0;
                        for (int j=0; j<b; ++j) {  //count hydrogens for while test
                                if (bonds[j]->isA(H))
                                        hydrogens++;
                        }
                        break;
                }
        }
    }
}

int Molecule::calcChainDir(FLOAT*& out, FLOAT* centers, int s, int d) {
    if (atoms.size()>100) {
        return 0;
    }
    atomT* silane=parent->parent->aTypes[8];
    atomT* ene=parent->parent->aTypes[9];
    atomT* methyl=parent->parent->aTypes[11];
    vector<Atom*> chain;
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)->type==silane) {
                chain.push_back(*i);
                break;
        }
    }
    if (chain.size()==0)
        return 0;
    int length=0;
    while (chain.back()->type!=methyl) {
        vector<Atom*> bs;
        int B=chain.back()->getBonded(bs);
        for (int i=0; i<B; ++i) {
                if (chain.size()>1) {
                        if (bs[i]==chain[chain.size()-2]) continue;
                }
                if (bs[i]->type==ene || bs[i]->type==methyl) {
                        //cout << bs[i] << endl;
                        chain.push_back(bs[i]);
                        ++length;
                        break;
```

```
                }
        }
    }
    length=chain.size();
    int l=length-1;
    #ifdef ipp
        FLOAT* rel;
        rel=IPPF1(ippsMalloc)(3*d*l);
        //rel2=IPPF1(ippsMalloc)(3*d);
    #else
        FLOAT rel[3*d*(length-1)];
    #endif
    relVecs(chain,rel,s,d);
    normalizeVecs(rel,l*d);
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        FLOAT* nPosToCent=IPPF1(ippsMalloc)(3*d*l);
        if (centers!=NULL) {
                FLOAT* posToCent=IPPF1(ippsMalloc)(3*d*l);
                FLOAT* pPosToCent[l];
                //need the relation vector from the first atom in each bonded pair to the center,
    so need these positions
                FLOAT* aPos[l];
                for (int i=0; i<l; ++i) {
                        aPos[i]=chain[i]->getPos(s);
                        pPosToCent[i]=posToCent+3*d*i;
                }
                IPPF2(ippmSub_vav,L)(aPos,0,stride2,centers,stride2,pPosToCent,0,stride2,3*d,l);
                FLOAT* norms=IPPF1(ippsMalloc)(d*l);
                IPPF1(ippmL2Norm_va)(posToCent,stride0,stride2,norms,3,d*l);
                FLOAT* inorms=IPPF1(ippsMalloc)(d*l);
                IPPF2(ippsInv,A53)(norms,inorms,d*l);

        IPPF1(ippmMul_vaca)(posToCent,stride0,stride2,inorms,stride2,nPosToCent,stride0,stride2,3
    ,d*l);
                ippsFree(posToCent); ippsFree(norms); ippsFree(inorms);
        } else {
                for (int i=0; i<3*d*l; ++i) {
                        if (i%3==2)
                                nPosToCent[i]=1;
                    else
                        nPosToCent[i]=0;
                }
        }
        FLOAT* dots=IPPF1(ippsMalloc)(d*l);
        out=IPPF1(ippsMalloc)(d*l);
        for (int i=0; i<l; i++)

        IPPF1(ippmDotProduct_vava)(nPosToCent+3*d*i,stride0,stride2,rel+3*d*i,stride0,stride2,dot
    s+d*i,3,d);
        IPPF2(ippsAcos,A53)(dots,out,d*l);
        IPPF2(ippsMulC,I)(R2D,out,d*l);
        ippsFree(dots);ippsFree(nPosToCent);ippsFree(rel);
    #else
        //TODO write non ipp branch
    #endif
    return l;
}

int Molecule::calcDihedChain(FLOAT*& out, int s, int d) {
    //If thing is huge its not a molecular chain
    if (atoms.size()>100) {
        return 0;
    }
```

```
    atomT* silane=parent->parent->aTypes[8];
    atomT* ene=parent->parent->aTypes[9];
    atomT* methyl=parent->parent->aTypes[11];
    vector<Atom*> chain;
    //DEB(chainres)
    //char buf;
    //cin >> buf;
    //chain.reserve(20);
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)->type==silane) {
                chain.push_back(*i);
                break;
        }
    }
    //No silane found handler
    if (chain.size()==0)
        return 0;
    int length=0;
    while (chain.back()->type!=methyl) {
        vector<Atom*> bs;
        int B=chain.back()->getBonded(bs);
        for (int i=0; i<B; ++i) {
                if (chain.size()>1) {
                        if (bs[i]==chain[chain.size()-2]) continue;
                }
                if (bs[i]->type==ene || bs[i]->type==methyl) {
                        //cout << bs[i] << endl;
                        chain.push_back(bs[i]);
                        ++length;
                        break;
                }
        }
    }
    length=chain.size();
    int l=length-3;
    #ifdef ipp
        FLOAT* rel;
        rel=IPPF1(ippsMalloc)(3*d*(length-1));
    #else
        //DEB(decREl)
        FLOAT rel[3*d*(length-1)];
    #endif
    relVecs(chain,rel,s,d);
    normalizeVecs(rel,(length-1)*d);
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        FLOAT* cr=IPPF1(ippsMalloc)((length-2)*3*d);
        for (int i=0; i<length-2; ++i)

        IPPF1(ippmCrossProduct_vava)(rel+3*d*i,stride0,stride2,rel+3*d*(i+1),stride0,stride2,cr+3
    *d*i,stride0,stride2,d);
        ippsFree(rel);
        FLOAT* crN=IPPF1(ippsMalloc)((length-2)*d);
        IPPF1(ippmL2Norm_va)(cr,stride0,stride2,crN,3,(length-2)*d);
        FLOAT* CR;
        for (int i=0; i<(length-2)*d; ++i) {
                CR=cr+3*i;
                for (int j=0; j<3; ++j) {
                        CR[j]=CR[j]/crN[i];
                }
        }
        FLOAT* dot=IPPF1(ippsMalloc)((length-3)*d);
        for (int i=0; i<length-3; ++i)
```

```
        IPPF1(ippmDotProduct_vava)(cr+3*d*i,stride0,stride2,cr+3*d*(i+1),stride0,stride2,dot+d*i,
    3,d);
        ippsFree(cr);
        out=IPPF1(ippsMalloc)((length-3)*d);
        IPPFIX(ippsAcos)(dot,out,(length-3)*d);
        IPPF2(ippsMulC,I)(R2D,out,(length-3)*d);
        ippsFree(dot);
    #else
        FLOAT cr[(length-2)*3*d];
        bigCrossProd(rel,length,d,cr);
        //DEB(decout)
        out=(FLOAT*)malloc(sizeof(FLOAT)*(length-3)*d);
        //DEB(aDecOut)
        //DEB(Dots and ACos)
        #pragma omp parallel for
        for (int i=0; i<d; ++i) {
                for (int j=0; j<length-3; ++j) {

                        out[l*i+j]=0;
                        for (int k=0; k<3; k++)
                                out[l*i+j]+=cr[3*d*j+3*i+k]*cr[3*d*(j+1)+3*i+k];
                        out[l*i+j]=acos(out[l*i+j])*R2D;
                }
        }
        ///DEB(Done)
    #endif
    return l;
    //DEB(done)
}

bool Molecule::leftRight(int dim, set<Atom*>& left, set<Atom*>& right) {
    FLOAT w=parent->parent->perDim[dim]/2.0;
    FLOAT l=parent->parent->dim[2*dim];
    bool split=0;
    FLOAT* pPos=(*atoms.begin())->pos;
    list<Atom*>::iterator j=atoms.begin(); ++j;
    while (!split) {
        FLOAT* tPos=(*j)->pos;
        if (fabs(tPos[dim]-pPos[dim])>w) {
                split=1;
                break;
        }
        pPos=tPos;
        ++j;
        if (j==atoms.end())
                break;
    }
    if (split) {
        for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
                FLOAT* pos=(*i)->pos;
                if ((pos[dim])>(l+w))
                        right.insert(*i);
                else
                        left.insert(*i);
        }
        return 1;
    }
    return 0;
}

bool Molecule::verify() {
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)->mol!=this)
                return 0;
    }
```

```
    return 1;
}

void Molecule::transfer(Molecule* inp) {
    while(!atoms.empty()) {
        inp->addAtom(*atoms.begin());
//      atoms.erase(atoms.begin());
    }
}

void Molecule::bound(FLOAT ang, FLOAT space) {
    FLOAT angRad=ang/R2D;  //Angle in radians
    FLOAT cut=acos(angRad);
    if (atoms.size()>200) {
        int t=parent->parent->aTypes[(*atoms.begin())->type];
        int* stat=(int*)malloc(sizeof(int)*atoms.size()); //0 means keep, 1 means delete, 2,
    means freeze
        list<Atom*>::iterator* its=new list<Atom*>::iterator[atoms.size()];
        if (t<8)  { //if its hte particle
                #ifndef ipp
                        #pragma omp parallel for
                        for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
                                FLOAT N,z;
                                FLOAT* aPos=(*i)->pos;
                                N=sqrt(pow(aPos[0],2)+pow(aPos[1],2)+pow(aPos[2],2));
                                z=aPos[2]/N;
                                if (z<cut) {
                                        FLOAT phi=acos(z);  //Angle of atom from z axis in
    radians
                                        FLOAT alph=angRad-phi;  //Angle between cutoff and atom
                                        FLOAT s=N*tan(alph); //Distance from atom to cutoff cone
                                        if (s<space) {
                                                stat[j]=2;
                                        } else {
                                                stat[j]=1;
                                        }
                                } else {
                                        stat[j]=0;
                                }
                        }
                #endif
                #ifdef ipp
                        int stride2=sizeof(FLOAT);
                        int stride0=3*stride2;
                        int j=0;
                        FLOAT** aPos=(FLOAT**)malloc(sizeof(FLOAT*)*atoms.size());
                        FLOAT* z=IPPF1(ippsMalloc)(atoms.size());
                        FLOAT* N=IPPF1(ippsMalloc)(atoms.size());
                        FLOAT* iN=IPPF1(ippsMalloc)(atoms.size());
                        for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i, ++j) {
                                its[j]=i;
                                aPos[j]=(*i)->pos;
                                z[j]=(*i)->pos[2];
                        }
                        IPPF2(ippmL2Norm_va,L)(aPos,0,stride2,N,3,atoms.size());
                        IPPFIX(ippsInv)(N,iN,atoms.size());
                        IPPF2(ippsMul,I)(iN,z,atoms.size());
                        FLOAT* phi=IPPF1(ippsMalloc)(atoms.size());
                        IPPFIX(ippsAcos)(z,phi,atoms.size());
                        FLOAT* alph=IPPF1(ippsMalloc)(atoms.size());
                        IPPF1(ippsSubC)(phi,angRad,alph,atoms.size());
                        IPPF2(ippsMulC,I)(-1,alph,atoms.size());
                        FLOAT* tn=IPPF1(ippsMalloc)(atoms.size());
                        IPPFIX(ippsTan)(alph,tn,atoms.size());
                        IPPF2(ippsMul,I)(N,tn,atoms.size());
```

```
                        #pragma omp parallel for
                        for (int i=0; i<atoms.size(); ++i) {
                                if (phi[i]<angRad)
                                        stat[i]=0;
                                else {
                                        if (tn[i]<space)
                                                stat[i]=2;
                                        else
                                                stat[i]=1;
                                }
                        }
                        ippsFree(z); ippsFree(N); ippsFree(iN);
                        ippsFree(phi); ippsFree(alph); ippsFree(tn); free(aPos);
                #endif
                atomT* HFrozen=new atomT(parent->parent->aTypes[7]->mass);
                HFrozen->eps=parent->parent->aTypes[7]->eps;
                HFrozen->sigma=parent->parent->aTypes[7]->sigma;
                atomT* OhFrozen=new atomT(parent->parent->aTypes[6]->mass);
                OhFrozen->eps=parent->parent->aTypes[6]->eps;
                OhFrozen->sigma=parent->parent->aTypes[6]->sigma;
                parent->parent->aTypes.addType(HFrozen,17);
                parent->parent->aTypes.addType(OhFrozen,16);
                vector<int> removals;
                #pragma omp parallel for
                for (int i=0; i<atoms.size(); ++i) {
                        if (stat[i]==0)
                                continue;
                        if (stat[i]==1)
                                #pragma omp critical
                                { removals.push_back((*its[i])->lIndex); }
                        if (stat[i]==2) {
                                int t=parent->parent->aTypes[(*its[i])->type];
                                switch (t)
                                {
                                        case 3:
                                                (*its[i])->type=parent->parent->aTypes[1];
                                                break;
                                        case 4:
                                                (*its[i])->type=parent->parent->aTypes[2];
                                                break;
                                        case 5:
                                                (*its[i])->type=parent->parent->aTypes[1];
                                                break;
                                        case 6:
                                                (*its[i])->type=parent->parent->aTypes[16];
                                                break;
                                        case 7:
                                                (*its[i])->type=parent->parent->aTypes[17];
                                                break;
                                }
                        }
                }
                parent->parent->atoms.remove(removals);
        } else {  //its the solvent
                //bye bye solvent molecules in angle phi---you are welcome, cbc
                for (int i=0; i<atoms.size(); ++i)
                        stat[i]=3;
                int j=0;
                for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i,++j) {
                        its[j]=i;
                }
                for (int i=0; i<atoms.size(); ++i) {
                        if (stat[i]==3) {

                        }
```

```
                }
        }
    } else { //its a silane


    }
}

void Molecule::clearAtoms() {
    while (!atoms.empty()) {
        parent->parent->atoms.remElem(*atoms.begin());
    }
}

void Molecule::translate(FLOAT* iPos) {
    FLOAT *tPos;
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
                tPos=(*i)->getPos();
                tPos[0]+=iPos[0];
                tPos[1]+=iPos[1];
                tPos[2]+=iPos[2];
    }
}

void Molecule::rotate(FLOAT* rot) {
    FLOAT *rotMat = IPPF1(ippsMalloc)(9);
    FLOAT *dot = IPPF1(ippsMalloc)(3);
    FLOAT *cross = IPPF1(ippsMalloc)(3);
    FLOAT dotR2,crossR2;
    dot[0] = 1, dot[1] = -rot[0]/rot[1]; dot[2] = 0;
    dotR2 = dot[0]*dot[0]+dot[1]*dot[1]+dot[2]*dot[2];
    int stride2 = sizeof(FLOAT);
    int stride1 = 3 * stride2;
    int stride0 = 3 * stride2;
    IPPF1(ippmCrossProduct_vv)(rot,stride2,dot,stride2,cross,stride2);
    FLOAT icn = 1.0/sqrt(cross[0]*cross[0]+cross[1]*cross[1]+cross[2]*cross[2]);
    FLOAT idn = 1.0/sqrt(1+rot[0]*rot[0]/(rot[1]*rot[1]));
    for (int i=0; i<3; ++i) {
        rotMat[3*i] = rot[i];
        rotMat[3*i+1] = dot[i]*idn;
        rotMat[3*i+2] = cross[i]*icn;
    }
    int n = atoms.size();
    FLOAT **aPos = (FLOAT**)malloc(sizeof(FLOAT*) * n);
    FLOAT **pNewPos = (FLOAT**)malloc(sizeof(FLOAT*) * n);
    FLOAT *newPos = IPPF1(ippsMalloc)(3 * n);
    int j=0;
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        aPos[j]=(*i)->getPos();
        pNewPos[j] = newPos+3*j;
        j++;
    }
    IPPF2(ippmMul_mva,L)(rotMat,stride1,stride2,3,3,aPos,0,stride2,3,pNewPos,0,stride2,n);
    for (int i=0; i<n; ++i) {
        aPos[i][0] = pNewPos[i][0];
        aPos[i][1] = pNewPos[i][1];
        aPos[i][2] = pNewPos[i][2];
    }
    ippsFree(rotMat); ippsFree(dot); ippsFree(cross);
    ippsFree(newPos);
    free(aPos);free(pNewPos);
}

void Molecule::findSilaneEnd(Atom* &silane, Atom* &firstC) {
    atomT* Si=parent->parent->aTypes[8];
```

```
    for (list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        if ((*i)->type==Si) {
                silane=(*i);
                vector<Atom*> bonds;
                silane->getBonded(bonds);
                firstC=bonds[0];
                return;
        }
    }
    DEB(Could not find silane or first carbon)
    return;
}
```

B.3 Element List Classes

Element list classes are used to contain the collections of atoms, bonds, etc. used in the simulation models. The memory handling and indexing features of these lists is maintained by a parent template class, 'ElemList', from which the individual list classes are derived.

Element List Class Template — **ElemList.h**

```
#ifndef ELEMLIST_H_
#define ELEMLIST_H_

class Particle;

template <typename T>
class ElemList
{
public:
    ElemList();
    virtual ~ElemList();

    unsigned int m,c;                   //m=memory size, c=element count
    queue<unsigned int> openInd; //list of available indices for which no element exists
    Particle* parent;                   //Parent particle that contains the list
    T** list;                           //List array, dynamically allocated list of pointers

    void reserve(unsigned int k=5000);  //Extend list size
    unsigned int size();                //returns c
    unsigned int max();                 //return m
    void compress();                    //Trims list so that no pointers are unallocated
    T* addElem(T*);                     //Adds element to the list via pointer
    void addElem(T**,int n);            //Adds a list of n elements to the list
    T* createNew();                     //Creates a new element in the list and returns a
    pointer to it
    void remElem(T*);                   //Removes element by its pointer (slow)
    void remElem(unsigned int);         //Removes element by its index
    T* operator[](unsigned int);        //returns pointer to element[i]
    void clear();                       //empties list
};

template <typename T>
void ElemList<T>::clear() {
    for (int i=0; i<m; ++i) {
```

```
        if (list[i]==NULL) continue;
        delete list[i];
        list[i]=NULL;
        --c;
        openInd.push(i);
    }
}

template <typename T>
ElemList<T>::ElemList<T>() {
    m=1; c=0;
    list=(T**)malloc(sizeof(T*));
    list[0]=NULL;
    openInd.push(0);
}

template <typename T>
ElemList<T>::~ElemList<T>() {
    for (int i=0; i<m; ++i) {
        if (list[i]!=NULL)
                delete list[i];
                list[i]=NULL;
    }
    free(list);
}

template <typename T>
T* ElemList<T>::addElem(T* t) {
    if (c==m) {
//      if (openInd.size()!=0) {
//              DEB(There are still open indices)
//              abort();
//      }
        reserve();
    }
    int ind=openInd.front();
    openInd.pop();
    list[ind]=t;
    list[ind]->parent=this;
    list[ind]->setInd(list+ind,ind);
    ++c;
    return t;
}

template <typename T>
T* ElemList<T>::createNew() {
    T* nw=new T();
    addElem(nw);
    return nw;
}

template <typename T>
void ElemList<T>::addElem(T** t, int n) {
    unsigned int T=n+c;
    if (m<T) {
        reserve(T-m);
    }
    for (int i=0; i<n; ++i) {
        if (t[i]!=NULL)
                addElem(t[i]);
    }
}

template <typename T>
unsigned int ElemList<T>::size() {
```

```
    return c;
}

template <typename T>
void ElemList<T>::remElem(T* t) {
    for (int i=0; i<m; ++i) {
        if (list[i]==t) {
                delete list[i];
                list[i]=NULL;
                openInd.push(i);
                --c;
                break;
        }
    }
}

template <typename T>
void ElemList<T>::reserve(unsigned int k) {
    if (k==0 && m==1 && c==0) {
        free(list);
        m=0;
    } else if (k==0)
        return;
    if (m==1 && c==0) {
        m=k;
        list=(T**)realloc(list,m*sizeof(T*));
        openInd.pop();
        for (int i=0; i<m; ++i) {
                list[i]=NULL;
                openInd.push(i);
        }
    } else {
        int ns=m+k;
        T** prev=list;
        list=(T**)realloc(list,ns*sizeof(T*));
        if (prev!=list) {
//              unsigned int j=0;
                for (int i=0; i<m; ++i) {
                        if (list[i]!=NULL) {
                                list[i]->setInd(list+i,i);
                                //++j;
                        }
                        else
                                list[i]=NULL;
                }
        }
        for (int i=m; i<ns; ++i) {
                list[i]=NULL;
                openInd.push(i);
        }
        m=ns;
    }
}

template <typename T>
T* ElemList<T>::operator[](unsigned int i) {
    return list[i];
}

template <typename T>
unsigned int ElemList<T>::max() {
    return m;
}

template <typename T>
```

```
void ElemList<T>::remElem(unsigned int k) {
    delete list[k];
    list[k]=NULL;
    openInd.push(k);
    --c;
}

template <typename T>
void ElemList<T>::compress() {
    if (m==c) return;
    T** nl=(T**)malloc(c*sizeof(T*));
    unsigned int j=0;
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                continue;
        nl[j]=list[i];
        nl[j]->setInd(nl+j,j);
        ++j;
    }
    free(list);
    list=nl;
    if (j>c) {
        DEB(Gained something during compress)
        abort();
    } else if (j<c) {
        DEB(Lost something during compress)
        abort();
    }
    m=c;
    while (!openInd.empty()) {
        openInd.pop();
    }
}

#endif /*ELEMLIST_H_*/
```

Atom List Class Declaration **AtomList.h**

```
#ifndef ATOMLIST_H_
#define ATOMLIST_H_

class Atom;

#include "restReader.h"
#include "data.h"

class AtomList : protected ElemList<Atom>
{
public:
    AtomList();
    virtual ~AtomList();

    //Inhereted members made public
    using ElemList<Atom>::parent;
    using ElemList<Atom>::size;
    using ElemList<Atom>::max;
    using ElemList<Atom>::operator[];
    using ElemList<Atom>::remElem;
    using ElemList<Atom>::clear;
    using ElemList<Atom>::compress;
    using ElemList<Atom>::createNew;
    using ElemList<Atom>::addElem;

    //Used for data output, maps printed indices to atom pointers
    map<unsigned int,Atom*> atmInd;
```

```
    map<unsigned int, unsigned int> atmOut;

    //Reading and writing data functions
    void readAtoms(istream&,int);
    void readAtoms(Data &data);
    void readAtomsNoMol(istream&,int,Molecule*);
    void writeAtomData(ostream&, int k=0);
    void writeAtomMol2(ostream&, int k=0);

    void fitDimension(FLOAT *dim, FLOAT* perDim);
    void deleteAtoms(Atom** rms);

    void setDumps(int,int,unsigned int*,FLOAT*); //reserves memory across all atoms for
    simulation snapshots
    void clearConnectivities();  //Removes all bonds, angles, etc from atoms in the list

    int dumped();  //number of atoms for which simulation snapshots exist

    void relVecs(unsigned int*,FLOAT*,int,int s=0,int d=0);      //Maybe redundant to global
    relvecs

    //functions for replicating periodic cell since lammps doesn't do this well when there's
    connectivity
    void replicate(int dim, int n, map<unsigned int,vector<Atom*> >& images);
    void replicate(int dim, int n, Molecule &m, Molecule* ms[], set<Atom*> &left, set<Atom*>
    &right, map<unsigned int,unsigned int* > &ax);

    void getDimLimits(int dim, FLOAT* out); //Gets boundaries based on atom positions
    void repos(FLOAT* dp);                                      //Shifts simulation cell

    //Transfers molecules from another particle to this one
    void transfer(Molecule &m, Molecule &newM, set<unsigned int> &bs, set<unsigned int> &as,
    set<unsigned int> &ds, set<unsigned int> &is, map<unsigned int, Atom*> &ax);
    void transAll(Particle &P, map<unsigned int,Atom*> &ax);

    Atom* findByFInd(unsigned int ind);  //Gets an atom by its file index

    void remove(vector<int> &rems);      //Removes atoms by their indices

    void calcPairEnergy(FLOAT* out, int s=0, int d=1);

    void rotate(FLOAT* r);               //Rotates using rotation matrix r
    void translate(FLOAT t[]);           //Translates by t
    void translatePer(FLOAT t[]);                //translate through periodic boundary conditions

    void cleave(FLOAT cut, int dim);     //Cleave atoms with dim position > cut
    void cleave(FLOAT *c);       //Cleaves atoms outside bounds specified c[0]->c[6] xlo xhi ylo
    etc...
    void cleaveCyl(FLOAT cut, int dim);  //removes atoms outside a cylinder of radius cut aligned
    along dimension dim
};

#endif /*ATOMLIST_H_*/
```

Atom list function definitions **AtomList.cpp**

```
#include "inc.h"
#include <float.h>


AtomList::AtomList()
{
}
```

```
AtomList::~AtomList()
{
}

void AtomList::fitDimension(FLOAT* dim, FLOAT* perDim) {
    FLOAT max[3]={DBL_MIN,DBL_MIN,DBL_MIN},min[3]={DBL_MAX,DBL_MAX,DBL_MAX};
    FLOAT *pos;
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        pos = list[i]->getPos();
        for (int j=0; j<3; ++j) {
                if (!parent->periodic[j]) {
                        if (max[j] < pos[j])
                                max[j] = pos[j];
                        if (min[j] > pos[j])
                                min[j] = pos[j];
                }
        }
    }
    if (!parent->periodic[0]) {
        dim[0] = min[0] - 5;
        dim[1] = max[0] + 5;
        perDim[0] = dim[1] - dim[0];
    }
    if (!parent->periodic[1]) {
        dim[2] = min[1] - 5;
        dim[3] = max[1] + 5;
        perDim[1] = dim[3] - dim[2];
    }
    if (!parent->periodic[2]) {
        dim[4] = min[2] - 5;
        dim[5] = max[2] + 5;
        perDim[2] = dim[5] - dim[4];
    }
}

void AtomList::readAtoms(istream& in, int k) {
    if (!parent->molecular) {
        Molecule *nm = new Molecule;
        parent->molecules.addElem(nm);
        readAtomsNoMol(in,k,nm);
        return;
    }
    reserve(k);
    map<int,Molecule*> molInd;
    for (unsigned int i=0; i<k; ++i) {
        if (i==(k-1))
                DEB(Stopping)
        FLOAT ch, pos[3]; unsigned int index; int mol, type, n[3];
        in >> index >> mol >> type >> ch >> pos[0] >> pos[1] >> pos[2] >> n[0] >> n[1] >> n[2];
        for (int d=0; d<3; d++) {
                if (parent->periodic[d]) {
                        while (pos[d]<parent->dim[2*d]) {
                                pos[d]+=parent->perDim[d];
                        }
                        while (pos[d]>parent->dim[2*d+1]) {
                                pos[d]-=parent->perDim[d];
                        }
                }
        }
        Atom* a=new Atom(pos,ch,parent->aTypes[type]);
        atmInd[index]=addElem(a);
        if (molInd.find(mol)==molInd.end()) {
                Molecule* moly=new Molecule();
                molInd[mol]=moly;
```

```
                parent->molecules.addElem(moly);
        }
        molInd[mol]->addAtom(a);
    }
}

void AtomList::readAtoms(Data &data) {
    reserve(data.natoms);
    map<int,Molecule*> molInd;
    Atom* ja;
    int index;
    for (int i=0; i<data.natoms; ++i) {
        index = data.tag[i];
        ja = addElem(new Atom(data.x[i],data.y[i],data.z[i],data.q[i],parent-
    >aTypes[data.type[i]]));
        for (int d=0; d<3; d++) {
                if (parent->periodic[d]) {
                        while (ja->pos[d]<parent->dim[2*d])
                                ja->pos[d]+=parent->perDim[d];
                        while (ja->pos[d]>parent->dim[2*d+1])
                                ja->pos[d]-=parent->perDim[d];
                }
        }
        atmInd[index] = ja;
        if (molInd.find(data.molecule[i])==molInd.end()) {
                parent->molecules.addElem(molInd[data.molecule[i]]=new Molecule());
        }
        molInd[data.molecule[i]]->addAtom(ja);
    }
}

void AtomList::readAtomsNoMol(istream& in, int k, Molecule* m) {
    reserve(k);
    float garb;
    for (unsigned int i=0; i<k; ++i) {
        FLOAT ch, pos[3]; unsigned int index; int type;
        in >> index >> type >> ch >> pos[0] >> pos[1] >> pos[2] >> garb >> garb >> garb;
        Atom* a=new Atom(pos,ch,parent->aTypes[type]);
        atmInd[index]=addElem(a);
        a->fileInd=index;
        m->addAtom(a);
    }
}

void AtomList::writeAtomData(ostream& out, int k) {
    atmOut.clear();
    compress();
    map<int, int> molInd;
    out << "Atoms" << endl << endl;
    unsigned int j=1;
    for (unsigned int i=0; i<m; ++i, ++j) {
        if (list[i]==NULL)
                continue;
        out << setw(15) << left << j;
        list[i]->writeData(out,molInd,j,k);
    }
    out << endl;
}

void AtomList::writeAtomMol2(ostream& out, int k) {
    atmOut.clear();
    map<int, int> molInd;
    elements el;
    out << "@<TRIPOS>ATOM" << endl;
    unsigned int j=1;
```

```
    for (unsigned int i=0; i<m; ++i) {
        if (list[i]==NULL)
                continue;
        if (list[i]->dumps==0 && k!=0)
                continue;
        FLOAT *aPos = list[i]->getPos(k);
        out << setw(15) << left << j;
        list[i]->writeMol2(out,molInd,el,j,k);
        ++j;
    }
    out << endl;
}

void AtomList::setDumps(int pDumps, int dc, unsigned int indices[], FLOAT dumpPos[]) {
    #pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i<m; ++i) {
                if (list[i]==NULL) continue;
                list[i]->resDumps(c);
        }
        #pragma omp for
        for (int i=0; i<dc; ++i) {
                for (int j=0; j<size(); j++) {
                        atmInd[indices[dc*i+j]]->setPos(dumpPos+3*j+dc*i,pDumps+dc);

                }
        }
    }
}

void AtomList::replicate(int dim, int n, map<unsigned int,vector<Atom*> >& images) {
    compress();
    unsigned int iniCnt=c;
    reserve(c*n);
    for (int i=0; i<iniCnt; ++i) {
        FLOAT* pos=list[i]->getPos();
        FLOAT ch=list[i]->charge;
        atomT* at=list[i]->type;
        vector<Atom*> ims;
        ims.reserve(n);
        for (int j=0; j<n; ++j) {
                FLOAT np[3];
                for (int k=0; k<3; ++k) {
                        if (k==dim)
                                np[k]=pos[k]+j*parent->perDim[dim];
                        else
                                np[k]=pos[k];
                }
                Atom* a=new Atom(np,ch,at);
                ims.push_back(addElem(a));
        }
        images[list[i]->lIndex]=ims;
    }
}

void AtomList::getDimLimits(int dim, FLOAT* out) {
    out[0]=1000000000;
    out[1]=-100000000;
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                continue;
        if (list[i]->pos[dim]<out[0])
                out[0]=list[i]->pos[dim];
        if (list[i]->pos[dim]>out[1])
```

```
                out[1]=list[i]->pos[dim];
    }
}

void AtomList::repos(FLOAT* dp) {
    #pragma omp sections
    {
        #pragma omp section
        {
                for (int i=0; i<m; ++i) {
                        if (list[i]==NULL)
                                continue;
                        list[i]->pos[0]+=dp[0];
                }
                parent->dim[0]+=dp[0];
                parent->dim[1]+=dp[0];
        }
        #pragma omp section
        {
                for (int i=0; i<m; ++i) {
                        if (list[i]==NULL)
                                continue;
                        list[i]->pos[1]+=dp[1];
                }
                parent->dim[2]+=dp[1];
                parent->dim[3]+=dp[1];
        }
        #pragma omp section
        {
                for (int i=0; i<m; ++i) {
                        if (list[i]==NULL)
                                continue;
                        list[i]->pos[2]+=dp[2];
                }
                parent->dim[4]+=dp[2];
                parent->dim[5]+=dp[2];
        }
    }
}

void AtomList::transfer(Molecule &m, Molecule &newM, set<unsigned int> &bs, set<unsigned int>
    &as, set<unsigned int> &ds, set<unsigned int> &is, map<unsigned int, Atom*> &ax) {
    for (std::list<Atom*>::iterator i=m.atoms.begin(); i!=m.atoms.end(); ++i) {
        Atom* old=(*i);
        old->collectConn(bs,as,ds,is);
        Atom* nw=new Atom(old->pos,old->charge,parent->aTypes[m.parent->parent->aTypes[old-
    >type]]);
        newM.addAtom(nw);
        ax[old->lIndex]=addElem(nw);
    }
}

void AtomList::replicate(int dim, int n, Molecule &m, Molecule* ms[], set<Atom*> &left,
    set<Atom*> &right, map<unsigned int,unsigned int*> &ax) {
    for (std::list<Atom*>::iterator i=m.atoms.begin(); i!=m.atoms.end(); ++i) {
                ax[(*i)->lIndex]=(unsigned int*)malloc(n*sizeof(unsigned int));
    }
    if (left.size()==0 && right.size()==0) {
        //this case the molecule is not split
        for (std::list<Atom*>::iterator i=m.atoms.begin(); i!=m.atoms.end(); ++i) {
                for (int j=0; j<n; ++j) {
                        Atom* old=*i;
                        Atom* a = new Atom((*i)->pos,(*i)->charge,(*i)->type);
                        a->pos[dim]+=(j+1)*(parent->perDim[dim]);
                        ms[j]->addAtom(addElem(a));
```

```
                        ax[(*i)->lIndex][j]=a->lIndex;
                }
        }
    } else if (left.size()==0 || right.size()==0) {
        DEB(Oddity on the molecule split)
    } else {
//      int l=left.size();
//      int r=right.size();
        //atoms on left of cell are associated with the cell they are in
        for (set<Atom*>::iterator i=left.begin(); i!=left.end(); ++i) {
                for (int j=0; j<n; ++j) {
                        Atom* a = new Atom((*i)->pos,(*i)->charge,(*i)->type);
                        a->pos[dim]+=(j+1)*(parent->perDim[dim]);
                        ms[j]->addAtom(addElem(a));
                        ax[(*i)->lIndex][j]=a->lIndex;
                }
        }
        //right atoms are associated with the next cell
        for (set<Atom*>::iterator i=right.begin(); i!=right.end(); ++i) {
                //Create atoms in all but last cell, putting them in the next molecule
                for (int j=0; j<n-1; ++j) {
                        Atom* a = new Atom((*i)->pos,(*i)->charge,(*i)->type);
                        a->pos[dim]+=(j+1)*(parent->perDim[dim]);
                        ms[j+1]->addAtom(addElem(a));
                        ax[(*i)->lIndex][j]=a->lIndex;
                }
                //Create last cell atom, and associate it with initial molecule
                if (!m.remAtom(*i))
                        DEB(Cant remove from mol)
                ms[0]->addAtom(*i);
                Atom* a = new Atom((*i)->pos,(*i)->charge,(*i)->type);
                a->pos[dim]+=(n)*(parent->perDim[dim]);
                m.addAtom(addElem(a));
                ax[(*i)->lIndex][n-1]=a->lIndex;
        }
    }
}

Atom* AtomList::findByFInd(unsigned int k) {
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                continue;
        if (list[i]->fileInd==k)
                return list[i];
    }
    DEB(Cannot find atom by file index)
    abort();
    return NULL;
}

void AtomList::remove(vector<int> &rems) {
    for (int i=0; i<rems.size(); ++i)
        remElem(rems[i]);
}

void AtomList::rotate(FLOAT *r) {
    FLOAT *tmp = IPPF1(ippsMalloc)(3);
    int stride2=sizeof(FLOAT);
    int stride1=3*stride2;
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        FLOAT* pos=list[i]->getPos(); //pos[0]=x, pos[1]=y, pos[2]=z
        IPPF1(ippmMul_mv)(r,stride1,stride2,3,3,pos,stride2,3,tmp,stride2);
        pos[0] = tmp[0];
        pos[1] = tmp[1];
```

```
        pos[2] = tmp[2];
    }
}

void AtomList::clearConnectivities() {
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        list[i]->clearConn();
    }
}

int AtomList::dumped() {
    int r=0;
    #pragma omp parallel for reduction(+:r)
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        if (list[i]->dumps!=0) r++;
    }
    if (r!=0)
        return r;
    else
        return c;
}

void AtomList::deleteAtoms(Atom** rms) {
    int M=m;
    for (int i=0; i<m; ++i) {
        if (rms[i]!=NULL)
                remElem(rms[i]->lIndex);
    }
}

void AtomList::translatePer(FLOAT t[]) {
    int d;
    FLOAT *pos;
#pragma omp parallel for private(d,pos)
    for (int i=0; i<m; ++i) {
        if (list[i]!=NULL) {
                pos = list[i]->getPos();
                for (d = 0; d < 3; ++d) {
                        pos[d]+=t[d];
                        if (parent->periodic[d]) {
                                while (pos[d] < parent->dim[2*d])
                                        pos[d]+=parent->perDim[d];
                                while (pos[d] > parent->dim[2*d+1])
                                        pos[d]-=parent->perDim[d];
                        }
                }
        }
    }
}

void AtomList::transAll(Particle &P,map<unsigned int,Atom*> &ax) {
    for (int i=0; i<P.atoms.max(); ++i) {
        if (P.atoms[i]==NULL) continue;
        Atom* old=P.atoms[i];
        Atom* nw=new Atom(old->pos,old->charge,parent->aTypes[P.aTypes[old->type]]);
        ax[old->lIndex]=addElem(nw);
    }
}

void AtomList::cleaveCyl(FLOAT c, int d) {
    Atom **rms = (Atom**)malloc(sizeof(Atom*)*m);
    int removals = 0;
    FLOAT *pos,rsq;
```

```
    FLOAT rMax2 = c*c;
    #pragma omp parallel for reduction(+:removals) private(pos)
    for (int i=0; i<m; ++i) {
        if (list[i]!=NULL) {
                pos = list[i]->getPos();
                rsq = pos[0]*pos[0] + pos[1]*pos[1] + pos[2]*pos[2];
                rsq -= pos[d]*pos[d];
                if (rsq>rMax2) {
                        rms[i] = list[i];
                        removals++;
                } else
                        rms[i]=NULL;
        } else
                rms[i]=NULL;
    }
    cout << "Removing " << removals << " atoms " << endl;
    deleteAtoms(rms);
    free(rms);
}

void AtomList::cleave(FLOAT c,int d) {
    Atom **rms=(Atom**)malloc(sizeof(Atom*)*m);
    int removals=0;
    #pragma omp parallel for reduction(+:removals)
    for (int i=0; i<m; ++i) {
        if (list[i]!=NULL) {
                //FLOAT* p=list[i]->getPos();
                if ((list[i]->getPos())[d]>c || (list[i]->getPos())[d]<(-c)) {
                        rms[i]=list[i];
                        removals++;
                } else
                        rms[i]=NULL;
        } else {
                rms[i]=NULL;
        }
    }
    cout << "Removing " << removals << " atoms" << endl;
    deleteAtoms(rms);
    free(rms);
}

void AtomList::cleave(FLOAT *c) {
    Atom **rms=(Atom**)malloc(sizeof(Atom*)*m);
    int removals = 0;
    FLOAT *p;
    #pragma omp parallel for reduction(+:removals) private(p)
    for (int i = 0; i<m; ++i) {
        if (list[i]==NULL) { rms[i]=NULL; continue; }
        p=list[i]->getPos();
        if (p[0]<c[0] || p[0]>c[1] || p[1]<c[2] || p[1]>c[3] || p[2]<c[4] || p[2]>c[5]) {
                rms[i]=list[i];
                removals++;
        } else
          rms[i]=NULL;
      }
    int r=0;
    int M=m;
    for (int i=0; i<M; ++i) {
      if(rms[i]!=NULL) {
        remElem(rms[i]->lIndex);
        r++;
      }
    }
    free(rms);
}
```

```
void AtomList::translate(FLOAT t[]) {
    #pragma omp parallel for
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        list[i]->pos[0]+=t[0];
        list[i]->pos[1]+=t[1];
        list[i]->pos[2]+=t[2];
    }
}
```

Bond list class declaration **BondList.h**

```
#ifndef BONDLIST_H_
#define BONDLIST_H_

#include "restReader.h"
#include "data.h"

class Bond;

class BondList : protected ElemList<Bond>
{
public:
    BondList();
    virtual ~BondList();

    //Inhereted from template and made public
    using ElemList<Bond>::parent;
    using ElemList<Bond>::size;
    using ElemList<Bond>::max;
    using ElemList<Bond>::remElem;
    using ElemList<Bond>::clear;
    using ElemList<Bond>::operator[];
    using ElemList<Bond>::compress;

    //File input and output functions
    void readBonds(istream&, int k);
    void readBonds(Data &data);
    void writeBondsData(ostream&);
    void writeBondsMol2(ostream&, int k=0);

    //Counts bonds that don't cross periodic boundary, used for mol2 file output
    int nonBndCount(int k=0);

    //Periodic replication functions
    void replicate(int dim, map<unsigned int, vector<Atom*> >& images);
    void replicate(int dim, int n, set<unsigned int> &bs, map<unsigned int,unsigned int*> &ax);
    void transfer(Molecule &m, set<unsigned int> &b, map<unsigned int,Atom*> &ax);
    void transAll(Particle &P, map<unsigned int,Atom*> &ax);

    //Resets all bonds, registering them with their atoms
    void resetConn();

    //Asks all bonds to verify, reports bonds that span 2 different molecules
    bool verify();

    //Creates a new bond between atoms with given type
    Bond* addBond(Atom* a1, Atom* a2, bondT *type);
};

#endif /*BONDLIST_H_*/
```

Bond list function definitions **BondList.cpp**

```
#include "inc.h"

BondList::BondList()
{
}

BondList::~BondList()
{
}

void BondList::readBonds(istream& in, int k) {
    reserve(k);
    unsigned int index, a1, a2; int t;
    for (int i=0; i<k; ++i) {
        in >> index >> t >> a1 >> a2;
        if (t==0) continue;
        Bond* b=new Bond(parent->atoms.atmInd[a1],parent->atoms.atmInd[a2],parent->bTypes[t]);
        addElem(b);
    }
}

void BondList::readBonds(Data &data) {
    reserve(data.nbonds);
    int a1,a2,bt;
    for (int i=0; i<data.nbonds; ++i) {
        bt = data.bond_type[i];
        if (bt==0) continue;
        a1 = data.bond_atom1[i];
        a2 = data.bond_atom2[i];
        addElem(new Bond(parent->atoms.atmInd[a1],parent->atoms.atmInd[a2],parent->bTypes[bt]));
    }
}

void BondList::writeBondsData(ostream& out) {
    if (c == 0) return;
    out << "Bonds" << endl << endl;
    unsigned int j=1;
    for (unsigned int i=0; i<m; ++i, ++j) {
        if (list[i]==NULL) continue;
        out << setw(15) << left << j;
        list[i]->writeBondData(out);
    }
    out << endl;
}

void BondList::writeBondsMol2(ostream& out, int k) {
    out << "@<TRIPOS>BOND" << endl;
    unsigned int j=1;
    Atom *l, *r;
    for (unsigned int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        if ((list[i]->atoms[0]->dumps==0 || list[i]->atoms[1]->dumps==0) && k!=0) continue;
        if (list[i]->isBnd(3,l,r,k)) {
                //cout << "m2 skipping bond" << endl;
                continue;
        }
        out << setw(15) << left << j;
        list[i]->writeBondMol2(out);
        ++j;
    }
    out << endl;
}
```

```
int BondList::nonBndCount(int k) {
    int counter=c;
    Atom* l,*r;
    for (int i=0; i<m; ++i) {
        if (list[i]!=NULL) {
                if ((list[i]->atoms[0]->dumps==0 || list[i]->atoms[1]->dumps==0) && k!=0)
                        --counter;
                if (list[i]->isBnd(3,l,r,k))
                        --counter;
        }
    }
    return counter;
}

void BondList::replicate(int dim, map<unsigned int, vector<Atom*> >& images) {
    compress();
    int n=images.begin()->second.size();
    int N=n+1;
    Bond* newBnds[c*N];
    unsigned int iniC=c;
    //#pragma omp parallel for
    for (unsigned int i=0; i<iniC; ++i) {
        Bond* b=list[i];
        bondT* bt=list[i]->type;
        Atom* a1=list[i]->atoms[0];
        Atom* a2=list[i]->atoms[1];
        Atom* l,*r;
        bool boundary=list[i]->isBnd(dim,l,r);
        if (!boundary) {
                for (int j=0; j<n; ++j)
                        newBnds[i*N+j]=new Bond(images[a1->lIndex][j],images[a2->lIndex][j],bt);
                //remElem(i);
                //newBnds[i*N+n]=new Bond(a1,a2,bt);
                newBnds[i*N+n]=NULL;
        } else {
                //This part skipped if n=1
                for (int j=0; j<n-1; ++j)
                        newBnds[i*N+j]=new Bond(images[r->lIndex][j],images[l->lIndex][j],bt);
                newBnds[i*N+n-1]=new Bond(images[r->lIndex][n-1],l,bt);
                remElem(i);
                newBnds[i*N+n]=new Bond(l,images[r->lIndex][0],bt);
        }
    }
    addElem(newBnds,c*N);
}

void BondList::transfer(Molecule &m, set<unsigned int> &b, map<unsigned int,Atom*> &ax) {
    int j=0;
    for (set<unsigned int>::iterator i=b.begin(); i!=b.end(); ++i) {
        int K=*i;
        Bond* B=m.parent->parent->bonds[K];
        Atom* a1=ax[B->atoms[0]->lIndex];
        Atom* a2=ax[B->atoms[1]->lIndex];
        bondT* type=parent->bTypes[m.parent->parent->bTypes[m.parent->parent->bonds[*i]->type]];
        Bond* bnd=new Bond(a1,a2,type);
        addElem(bnd);
        ++j;
    }
}

void BondList::replicate(int dim, int n, set<unsigned int> &bs, map<unsigned int,unsigned int* >
    &ax) {
    Atom *l, *r;
    //cout << "Processing " << bs.size() << " bonds" << endl;
```

```
    for (set<unsigned int>::iterator i=bs.begin(); i!=bs.end(); ++i) {
        Bond* tBond=list[*i];
        unsigned int oa1=tBond->atoms[0]->lIndex;
        unsigned int oa2=tBond->atoms[1]->lIndex;
        if (tBond->isBnd(dim,l,r)) {
                //cout << "split" << endl;
                for (int j=1; j<n; ++j) {
                        Atom* na1=parent->atoms[ax[l->lIndex][j]];
                        Atom* na2=parent->atoms[ax[r->lIndex][j-1]];
                        Bond* newB = new Bond(na1,na2,tBond->type);
                        addElem(newB);
                }
                bondT* type=tBond->type;
                remElem(tBond);
                tBond->verify();
                Atom* lastR=parent->atoms[ax[r->lIndex][n-1]]; //this is the rightmost atom in
    last cell
                Atom* firstL=parent->atoms[ax[l->lIndex][0]]; //this is left atom in first cell
                Bond* acrossBnd = new Bond(lastR,l,type);
                Bond* firstBnd = new Bond(r,firstL,type);
                addElem(acrossBnd);
                addElem(firstBnd);
        } else {
                //cout << "cont" << endl;
                for (int j=0; j<n; ++j) {
                        Atom* na1=parent->atoms[ax[oa1][j]];
                        Atom* na2=parent->atoms[ax[oa2][j]];
                        Bond* newB = new Bond(na1,na2,tBond->type);
                        addElem(newB);
                }
        }
    }
}

bool BondList::verify() {
    bool tests[m];
    #pragma omp parallel for
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                tests[i]=1;
        else
                tests[i]=list[i]->verify();
    }
    bool r;
    for (int i=0; i<m; ++i) {
        if (!tests[i]) {
                cout << "Bond " << i << " spans molecules ";
                cout << list[i]->atoms[0]->mol->lIndex << '(' << list[i]->atoms[0]->mol << ')';
                cout << ' ' << list[i]->atoms[1]->mol->lIndex << '(' << list[i]->atoms[1]->mol <<
    ')' << endl;
                r=0;
        }
    }
}

Bond* BondList::addBond(Atom* a1, Atom* a2, bondT *type) {
    Bond* newB=new Bond(a1,a2,type);
    addElem(newB);
}

void BondList::resetConn() {
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        list[i]->rereg();
    }
```

```
}

void BondList::transAll(Particle &P,map<unsigned int,Atom*> &ax) {
    Atom *a1,*a2;
    Bond *B,*nBnd;
    bondT* type;
    for (int i=0; i<P.bonds.max(); ++i) {
        if (P.bonds[i]==NULL) continue;
        B=P.bonds[i];
        a1=ax[B->atoms[0]->lIndex];
        a2=ax[B->atoms[1]->lIndex];
        type=parent->bTypes[P.bTypes[B->type]];
        nBnd=new Bond(a1,a2,type);
        addElem(nBnd);
    }
}
```

Angle list class declaration — AngleList.h

```
#ifndef ANGLELIST_H_
#define ANGLELIST_H_

#include "restReader.h"
#include "data.h"

class Angle;

class AngleList : protected ElemList<Angle>
{
public:
//Constructors & Destructors
    AngleList();
    virtual ~AngleList();

    //derived elements made public
    using ElemList<Angle>::parent;
    using ElemList<Angle>::size;
    using ElemList<Angle>::max;
    using ElemList<Angle>::remElem;
    using ElemList<Angle>::clear;
    using ElemList<Angle>::operator[];
    using ElemList<Angle>::compress;
    using ElemList<Angle>::addElem;

    //file input/output
    void readAngles(istream&,int); //reads angles form lammps format data file, cursor must be at
    first angle position
    void readAngles(Data &data);
    void writeAngleData(ostream&); //governs angle writing, does indexing and directs angle to
    write their info

    void parseForAlk(Molecule* m, vector<Angle*> &angs); //parses angles from a mol2 derived
    particle of alkanes
    void parseForSil(vector<Angle*> &angs); //parses angles from a mol2 derived alkylsilane
    molecule
    void parseFor3p1p(Molecule* m, vector<Angle*> &angs); //parses angles from mol2 derived 3-
    phenyl-1-propanol

    //Handles replications over periodic boundary conditions
    void replicate(int dim, map<unsigned int,vector<Atom*> >& images); /*replications replicate
    angles for cell replication*/
    void replicate(int dim, int n, set<unsigned int> &as, map<unsigned int,unsigned int*> &ax);
    void transfer(Molecule &m, set<unsigned int> &a, map<unsigned int,Atom*> &ax); //transfers
    angles from molecule m in another particle to this particle, assumes molecule has been
```

```
    transferred in, ax links the two
    void transAll(Particle &P, map<unsigned int,Atom*> &ax);

    bool verify(int k); //verifies angles are contained within molecules

    Angle* addAngle(Atom* a1, Atom* a2, Atom* a3, angleT* type); //creates a new angle using the
    info specified, returns pointer to said angle

    //Resets angles, reregistering with consituent atoms
    void resetConn();
};

#endif /*ANGLELIST_H_*/
```

Angle list function definitions **AngleList.cpp**

```
#include "inc.h"

AngleList::AngleList()
{
}

AngleList::~AngleList()
{
}

void AngleList::readAngles(istream& in,int k) {
    reserve(k-m+c);
    for (int i=0; i<k; ++i) {
        unsigned int index, a1, a2, a3; int type;
        in >> index >> type >> a1 >> a2 >> a3;
        if (type==0) continue;
        Angle* a=new Angle(parent->atoms.atmInd[a1],parent->atoms.atmInd[a2],parent-
    >atoms.atmInd[a3],parent->angTypes[type]);
        addElem(a);
    }
}

void AngleList::readAngles(Data &data) {
    reserve(data.nangles);
    int a1, a2, a3, at;
    for (int i=0; i<data.nangles; ++i) {
        at = data.angle_type[i];
        if (at==0) continue;
        a1 = data.angle_atom1[i];
        a2 = data.angle_atom2[i];
        a3 = data.angle_atom3[i];
        addElem(new Angle(
                parent->atoms.atmInd[a1],
                parent->atoms.atmInd[a2],
                parent->atoms.atmInd[a3],
                parent->angTypes[at]));
    }
}

void AngleList::writeAngleData(ostream& out) {
    if (c==0) return;
    out << "Angles" << endl << endl;
    unsigned int j=1;
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        out << setw(15) << left << j;
        list[i]->writeAngleData(out);
        ++j;
    }
```

```
    out << endl;
}

Angle* AngleList::addAngle(Atom* a1, Atom* a2, Atom* a3, angleT* type) {
    return addElem(new Angle(a1,a2,a3,type));
}

void AngleList::replicate(int dim, map<unsigned int,vector<Atom*> >& images) {
    compress();
    int n=images.begin()->second.size();
    Angle* newAngs[c*n];
    unsigned int iniC=c;
    //#pragma omp parallel for
    for (unsigned int i=0; i<iniC; ++i) {
        angleT* at=list[i]->type;
        set<short> l,r;
        Atom* iA[3]={list[i]->atoms[0], list[i]->atoms[1], list[i]->atoms[2]};
        bool bnd=list[i]->isBnd(dim,l,r);
        if (bnd==0) {
                for (int j=0; j<n; ++j) {
                        newAngs[i*n+j]=new Angle(images[iA[0]->lIndex][j],images[iA[1]-
    >lIndex][j],images[iA[2]->lIndex][j],at);
                }
        } else {
                for (int j=0; j<n-1; ++j) {
                        Angle* a=(newAngs[i*n+j]=new Angle());
                        a->type=at;
                        for (int k=0; k<3; ++k) {
                                if (r.find(k)!=r.end())
                                        a->atoms[k]=images[iA[k]->lIndex][j];
                                else
                                        a->atoms[k]=images[iA[k]->lIndex][j+1];
                                a->atoms[k]->addAngle(a);
                        }
                }
                Angle* a=(newAngs[i*n+n-1]=new Angle());
                a->type=at;
                for (int k=0; k<3; ++k) {
                        if (r.find(k)!=r.end())
                                a->atoms[k]=images[iA[k]->lIndex][n-1];
                        else
                                a->atoms[k]=iA[k];
                        a->atoms[k]->addAngle(a);
                }
                for (int k=0; k<3; ++k) {
                        if (l.find(k)!=l.end()) {
                                list[i]->atoms[k]->remAngle(list[i]);
                                list[i]->atoms[k]=images[iA[k]->lIndex][0];
                                list[i]->atoms[k]->addAngle(list[i]);
                        }
                }
        }
    }
    addElem(newAngs,c*n);
}

void AngleList::transfer(Molecule &m, set<unsigned int> &a, map<unsigned int,Atom*> &ax) {
    for (set<unsigned int>::iterator i=a.begin(); i!=a.end(); ++i) {
        Atom* a1=ax[m.parent->parent->angles[*i]->atoms[0]->lIndex];
        Atom* a2=ax[m.parent->parent->angles[*i]->atoms[1]->lIndex];
        Atom* a3=ax[m.parent->parent->angles[*i]->atoms[2]->lIndex];
        angleT* type=parent->angTypes[m.parent->parent->angTypes[m.parent->parent->angles[*i]-
    >type]];
        Angle* ang=new Angle(a1,a2,a3,type);
        addElem(ang);
```

```
    }
}

void AngleList::replicate(int dim, int n, set<unsigned int> &as, map<unsigned int,unsigned int* >
    &ax) {
    for (set<unsigned int>::iterator i=as.begin(); i!=as.end(); ++i) {
        set<short> l, r;
        Angle* tAng=list[*i];
        unsigned int oa[3]={tAng->atoms[0]->lIndex, tAng->atoms[1]->lIndex, tAng->atoms[2]-
    >lIndex};
        if (tAng->isBnd(dim,l,r)) {
                for (int j=0; j<n-1; ++j) {
                        Angle* a=new Angle();
                        a->type=tAng->type;
                        for (int k=0; k<3; ++k) {
                                if (r.find(k)==r.end())
                                        a->atoms[k]=parent->atoms[ax[oa[k]][j+1]];
                                else
                                        a->atoms[k]=parent->atoms[ax[oa[k]][j]];
                                a->atoms[k]->addAngle(a);
                        }
                        addElem(a);
                }
                angleT* type=tAng->type;
                remElem(tAng->lIndex);
                Atom* cell0[3], *lCell[3];
                for (int k=0; k<3; ++k) {
                        if (r.find(k)!=r.end()) {
                                cell0[k]=parent->atoms[ax[oa[k]][n-1]];
                                lCell[k]=parent->atoms[oa[k]];
                        } else {
                                cell0[k]=parent->atoms[oa[k]];
                                lCell[k]=parent->atoms[ax[oa[k]][0]];
                        }
                }
                Angle* n1 = new Angle(cell0[0], cell0[1], cell0[2], type);
                Angle* n2 = new Angle(lCell[0], lCell[1], lCell[2], type);
                addElem(n1); addElem(n2);
        } else {
                for (int j=0; j<n; ++j) {
                        Atom* na1=parent->atoms[ax[oa[0]][j]];
                        Atom* na2=parent->atoms[ax[oa[1]][j]];
                        Atom* na3=parent->atoms[ax[oa[2]][j]];
                        Angle* newA = new Angle(na1,na2,na3,tAng->type);
                        addElem(newA);
                }
        }
    }
}

bool AngleList::verify(int k) {
    bool tests[m];
    #pragma omp parallel for
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                tests[i]=1;
        else
                tests[i]=list[i]->verify(k);
    }
    bool r=1;
    for (int i=0; i<m; ++i) {
        if (!tests[i]) {
                cout << "Angle " << i << " failed verification" << endl;
                r=0;
        }
```

```
    }
    return r;
}

void AngleList::resetConn() {
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        list[i]->rereg();
    }
}

void AngleList::transAll(Particle &P,map<unsigned int,Atom*> &ax) {
    angleT* type;
    Angle *ang,*nAngle;
    Atom* a[3];
    for (int i=0; i<P.angles.max(); ++i) {
        if (P.angles[i]==NULL) continue;
        ang=P.angles[i];
        for (int j=0; j<3; ++j)
                a[j]=ax[ang->atoms[j]->lIndex];
        type=parent->angTypes[P.angTypes[ang->type]];
        nAngle=new Angle(a[0],a[1],a[2],type);
        addElem(nAngle);
    }
}

void AngleList::parseForSil(vector<Angle*>& out) {
    int b;
    atomT* t1,*t2,*t3; Atom *a1,*a2,*a3;
    atomT* H=parent->aTypes[12];
    atomT* C1=parent->aTypes[9];
    atomT* C2=parent->aTypes[11];
    atomT* Si=parent->aTypes[8];
    Angle* ang;
    int m=parent->atoms.max();
    for (int i=0; i<m; ++i) {
        if(parent->atoms[i]==NULL) continue;
        a2=parent->atoms[i];
        vector<Atom*> bonds;
        b=a2->getBonded(bonds);
        if (b==1) continue;
        for (int j=0; j<b; ++j) {
                a1=bonds[j];
                for (int k=j+1; k<b; ++k) {
                        a3=bonds[k];
                        t1=a1->type;
                        t2=a2->type;
                        t3=a3->type;
                        if ((t1==Si && t2==C1 && t3==C1) || (t3==Si && t2==C1 && t1==C1)) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[4]);
                        } else if (t1==H && (t2==C1 || t2==C2) && t3==H) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[9]);
                        } else if ((t1==C1 || t1==C2) && t2==C1 && (t3==C1 || t3==C2)) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[7]);
                        } else if ((t1==C1 || t1==C2) && (t2==C1 || t2==C2) && t3==H) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[8]);
                        } else if ((t3==C1 || t3==C2) && (t2==C1 || t2==C2) && t1==H) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[8]);
                        } else if ((t1==Si && t2==C1 && t3==H) || (t3==Si && t2==C1 && t1==H)) {
                                ang=new Angle(a1,a2,a3,parent->angTypes[5]);
                        } else {
                                DEB(ParseForSil Couldnt type angle)
                                cout << parent->aTypes[t1] << ' ' << parent->aTypes[t2] << ' '
    << parent->aTypes[t3] << endl;
                                continue;
```

```
                        }
                        out.push_back(addElem(ang));
                }
        }
    }
}
```

Dihedral list class definition **DihedList.h**

```
#ifndef DIHEDLIST_H_
#define DIHEDLIST_H_

#include "restReader.h"
#include "data.h"

class Dihed;

class DihedList : protected ElemList<Dihed>
{
public:
//Constructors & Destructors
    DihedList();
    ~DihedList();

    //Derived elements in use
    using ElemList<Dihed>::parent;
    using ElemList<Dihed>::size;
    using ElemList<Dihed>::max;
    using ElemList<Dihed>::remElem;
    using ElemList<Dihed>::clear;
    using ElemList<Dihed>::operator[];
    using ElemList<Dihed>::compress;

    //file input/output
    void readDihedData(istream&,unsigned int); //reads dihedral data from Lammps format data
    file, cursor should be positioned at first dihedral index
    void readDihedData(Data &data);
    void writeDihedData(ostream&); //governs writing of dihedral data, primarily dictated to
    invididual dihedrals

    void parseForAlk(vector<Angle*> &angs); //parses dihedrals out of alkane system, for
    solvating particle in hexane solvent
    void parseForSil(vector<Angle*> &angs); //parses dihedrals for alkylsilane from read in mol2
    file
    void parseFor3p1p(vector<Angle*> &angs); //parses dihedrals in 3p1p

    //Replication over period boundary condition functions
    void replicate(int dim, map<unsigned int, vector<Atom*> >& images); /*replicate are functions
    that transpose base cell into replica cells*/
    void replicate(int dim, int n, set<unsigned int> &ds, map<unsigned int,unsigned int*> &ax);
    void transfer(Molecule &m, set<unsigned int> &d, map<unsigned int,Atom*> &ax); //transers
    dihedrals from molecule m from another particle to this particle, ax links the atoms together
    void transAll(Particle &P, map<unsigned int,Atom*> &ax);

    bool verify(int k); //verifies dihedrals contained within molecules

    void resetConn();
};

#endif /*DIHEDLIST_H_*/
```

Dihedral list function definitions **DihedList.cpp**

```
#include "inc.h"

DihedList::DihedList()
{
}

DihedList::~DihedList()
{
}

void DihedList::readDihedData(istream& in, unsigned int k) {
    if (k==0) return;
    reserve(k);
    for (unsigned int i=0; i<k; ++i) {
        unsigned int a1, a2, a3, a4, index; int type;
        in >> index >> type >> a1 >> a2 >> a3 >> a4;
        if (type==0) continue;
        Dihed* d=new Dihed(parent->atoms.atmInd[a1],parent->atoms.atmInd[a2],parent-
    >atoms.atmInd[a3],parent->atoms.atmInd[a4],parent->dTypes[type]);
        addElem(d);
    }
}

void DihedList::readDihedData(Data &data) {
    reserve(data.ndihedrals);
    int a1, a2, a3, a4, dt;
    for (int i=0; i<data.ndihedrals; ++i) {
        dt = data.dihedral_type[i];
        if (dt==0) continue;
        a1 = data.dihedral_atom1[i];
        a2 = data.dihedral_atom2[i];
        a3 = data.dihedral_atom3[i];
        a4 = data.dihedral_atom4[i];
        addElem(new Dihed(
                parent->atoms.atmInd[a1],
                parent->atoms.atmInd[a2],
                parent->atoms.atmInd[a3],
                parent->atoms.atmInd[a4],
                parent->dTypes[dt]));
    }
}

void DihedList::writeDihedData(ostream& out) {
    if (m==0) return;
    out << "Dihedrals" << endl << endl;
    unsigned int j=1;
    for (unsigned int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        out << setw(15) << left << j;
        list[i]->writeDihData(out);
        ++j;
    }
    out << endl;
}

void DihedList::replicate(int dim, map<unsigned int,vector<Atom*> >& images) {
    compress();
    unsigned int iniC=c;
    int n=images.begin()->second.size();
    Dihed* newDihs[c*n];
    for (unsigned int i=0; i<iniC; ++i) {
        dihedT* dt=list[i]->type;
        Atom* iA[4]={list[i]->atoms[0],list[i]->atoms[1],list[i]->atoms[2],list[i]->atoms[3]};
        set<short> l,r;
```

```
        bool bnd=list[i]->leftRight(dim,l,r);
        if (bnd==0) {
                for (int j=0; j<n; ++j) {
                        newDihs[i*n+j]=new Dihed(images[iA[0]->lIndex][j],images[iA[1]-
    >lIndex][j],images[iA[2]->lIndex][j],images[iA[3]->lIndex][j],dt);
                }
        } else {
                //Create boundary dihedrals on right edge of new cells
                for (int j=0; j<n-1; ++j) {
                        Dihed* d=(newDihs[i*n+j]=new Dihed());
                        d->type=dt;
                        for (int k=0; k<4; ++k) {
                                if (r.find(k)!=r.end())
                                        d->atoms[k]=images[iA[k]->lIndex][j];
                                else
                                        d->atoms[k]=images[iA[k]->lIndex][j+1];
                                d->atoms[k]->addDih(d);
                        }
                }
                //Create boundary dihedrals on right edge of last cell looped back to initial
    cell
                Dihed* d=(newDihs[i*n+n-1]=new Dihed());
                d->type=dt;
                for (int k=0; k<4; ++k) {
                        if (r.find(k)!=r.end())
                                d->atoms[k]=images[iA[k]->lIndex][n-1];
                        else
                                d->atoms[k]=iA[k];
                        d->atoms[k]->addDih(d);
                }
                //Reconnect dihedrals on right of initial cell to first replicate
                for (int k=0; k<4; ++k) {
                        if (l.find(k)!=l.end()) {
                                list[i]->atoms[k]->remDih(list[i]);
                                list[i]->atoms[k]=images[iA[k]->lIndex][0];
                                list[i]->atoms[k]->addDih(list[i]);
                        }
                }
        }
    }
    addElem(newDihs,c*n);
}

void DihedList::transfer(Molecule &m, set<unsigned int> &d, map<unsigned int, Atom*> &ax) {
    for (set<unsigned int>::iterator i=d.begin(); i!=d.end(); ++i) {
        Atom* a1=ax[m.parent->parent->dihedrals[*i]->atoms[0]->lIndex];
        Atom* a2=ax[m.parent->parent->dihedrals[*i]->atoms[1]->lIndex];
        Atom* a3=ax[m.parent->parent->dihedrals[*i]->atoms[2]->lIndex];
        Atom* a4=ax[m.parent->parent->dihedrals[*i]->atoms[3]->lIndex];
        dihedT* type=parent->dTypes[m.parent->parent->dTypes[m.parent->parent->dihedrals[*i]-
    >type]];
        Dihed* dih=new Dihed(a1,a2,a3,a4,type);
        addElem(dih);
    }
}

void DihedList::replicate(int dim, int n, set<unsigned int> &ds, map<unsigned int,unsigned int*>
    &ax) {
    for (set<unsigned int>::iterator i=ds.begin(); i!=ds.end(); ++i) {
        set<short> l, r;
        Dihed* tDih=list[*i];
        unsigned int oa[4]={tDih->atoms[0]->lIndex, tDih->atoms[1]->lIndex, tDih->atoms[2]-
    >lIndex, tDih->atoms[3]->lIndex};
        if (tDih->leftRight(dim,l,r)) {
                for (int j=0; j<n-1; ++j) {
```

```
                        Dihed* d=new Dihed();
                        d->type=tDih->type;
                        for (int k=0; k<4; ++k) {
                                if (r.find(k)==r.end())
                                        d->atoms[k]=parent->atoms[ax[oa[k]][j+1]];
                                else
                                        d->atoms[k]=parent->atoms[ax[oa[k]][j]];
                                d->atoms[k]->addDih(d);
                        }
                        addElem(d);
                }
                dihedT* type=tDih->type;
                remElem(tDih->lIndex);
                Atom* cell0[4], *lCell[4];
                for (int k=0; k<4; ++k) {
                        if (r.find(k)!=r.end()) {
                                cell0[k]=parent->atoms[ax[oa[k]][n-1]];
                                lCell[k]=parent->atoms[oa[k]];
                        } else {
                                cell0[k]=parent->atoms[oa[k]];
                                lCell[k]=parent->atoms[ax[oa[k]][0]];
                        }
                }
                Dihed* d1 = new Dihed(cell0[0],cell0[1],cell0[2],cell0[3],type);
                Dihed* d2 = new Dihed(lCell[0],lCell[1],lCell[2],lCell[3],type);
                addElem(d1); addElem(d2);
        } else {
                for (int j=0; j<n; ++j) {
                        Atom* a1=parent->atoms[ax[oa[0]][j]];
                        Atom* a2=parent->atoms[ax[oa[1]][j]];
                        Atom* a3=parent->atoms[ax[oa[2]][j]];
                        Atom* a4=parent->atoms[ax[oa[3]][j]];
                        Dihed* d = new Dihed(a1,a2,a3,a4,tDih->type);
                        addElem(d);
                }
        }
    }
}

bool DihedList::verify(int k) {
    bool tests[m];
    #pragma omp parallel for
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL)
                tests[i]=1;
        else
                tests[i]=list[i]->verify(k);
    }
    bool r=1;
    for (int i=0; i<m; ++i) {
        if (!tests[i]) {
                cout << "Dihedral " << i << " failed verification" << endl;
                r=0;
        }
    }
    return r;
}

void DihedList::resetConn() {
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) continue;
        list[i]->rereg();
    }
}
```

```
void DihedList::transAll(Particle &P,map<unsigned int,Atom*> &ax) {
    dihedT* type;
    Dihed *dih,*nDih;
    Atom* a[4];
    for (int i=0; i<P.dihedrals.max(); ++i) {
        if (P.dihedrals[i]==NULL) continue;
        dih=P.dihedrals[i];
        for (int j=0; j<4; ++j)
                a[j]=ax[dih->atoms[j]->lIndex];
        type=parent->dTypes[P.dTypes[dih->type]];
        nDih=new Dihed(a[0],a[1],a[2],a[3],type);
        addElem(nDih);
    }
}

void DihedList::parseForSil(vector<Angle*> &angs) {
    Dihed* newD;
    atomT *t1, *t2, *t3, *t4;
    atomT* H=parent->aTypes[12];
    atomT* C1=parent->aTypes[9];
    atomT* C2=parent->aTypes[11];
    dihedT* CCCC=parent->dTypes[1];
    dihedT* HCCC=parent->dTypes[2];
    dihedT* HCCH=parent->dTypes[3];
    for (int i=0; i<angs.size(); ++i) {
        for (int j=i; j<angs.size(); ++j) {
                newD=(*angs[i])+(*angs[j]);
                if (newD==NULL) continue;
                t1=newD->atoms[0]->type;
                t2=newD->atoms[1]->type;
                t3=newD->atoms[2]->type;
                t4=newD->atoms[3]->type;
                if ((t1==C1 || t1==C2) && t2==C1 && t3==C1 && (t4==C1 || t4==C2)) {
                        newD->type=CCCC;
                        addElem(newD);
                } else if (t1==H && (t2==C1 || t2==C2) && t3==C1 && (t4==C1 || t4==C2)) {
                        newD->type=HCCC;
                        addElem(newD);
                } else if ((t1==C1 || t1==C2) && t2==C1 && (t3==C1 || t3==C2) && t4==H) {
                        newD->type=HCCC;
                        addElem(newD);
                } else if (t1==H && t4==H && (t2==C1 || t2==C2) && (t3==C1 || t3==C2)) {
                        newD->type=HCCH;
                        addElem(newD);
                } else {
                        DEB(ParseForSil Couldnt type dihedral)
                        delete newD;
                        cout << parent->aTypes[t1] << ' ' << parent->aTypes[t2] << ' ' <<
    parent->aTypes[t3] << ' ' << parent->aTypes[t4] << endl;
                        continue;
                }
        }
    }
}

void ImpList::transAll(Particle &P,map<unsigned int,Atom*> &ax) {
    impT* type;
    Improper *imp,*nImp;
    Atom* a[4];
    for (int i=0; i<P.impropers.max(); ++i) {
        if (P.impropers[i]==NULL) continue;
        imp=P.impropers[i];
        for (int j=0; j<4; ++j)
                a[j]=ax[imp->atoms[j]->lIndex];
        type=parent->iTypes[P.iTypes[imp->type]];
```

```
        nImp=new Improper(a[0],a[1],a[2],a[3],type);
        addElem(nImp);
    }
}
```

Molecule list class definition — MolecList.h

```
#ifndef MOLECLIST_H_
#define MOLECLIST_H_

class Molecule;

class MolecList : protected ElemList<Molecule>
{
public:
    //constructors/destructors
    MolecList();
    virtual ~MolecList();

    //Inhereted elements made public
    using ElemList<Molecule>::parent;
    using ElemList<Molecule>::addElem;
    using ElemList<Molecule>::operator[];
    using ElemList<Molecule>::size;
    using ElemList<Molecule>::max;
    using ElemList<Molecule>::compress;
    using ElemList<Molecule>::clear;
    using ElemList<Molecule>::remElem;

    //Replication of periodic cells
    void replicate(int dim, map<unsigned int,vector<Atom*> >& images);
    void replicate(int dim, int n);

    void detCon(); //Converts atom sets into individual molecules

    void readAlk(istream&); //reads input alkane solvent molecule
    void read3P1P(istream&);     //reads input 3p1p solvent molecule
    void combine(vector<Molecule*> toc); //Combines list of molecules into one
    void split(int tbs, vector<Molecule*> &out);

    bool verify();

    //Transfers all molecules from particle P into this particle
    void transAll(Particle &P, map<unsigned int,Atom*> &ax);
};

#endif /*MOLECLIST_H_*/
```

Molecule list function definitions — MolecList.cpp

```
#include <sstream>
#include "inc.h"

MolecList::MolecList()
{
    reserve(20);
}

MolecList::~MolecList()
{
}
```

```
void MolecList::detCon() {
    for (int i=0; i<parent->atoms.max(); ++i) {
        if (parent->atoms[i]==NULL)
                continue;
        if (parent->atoms[i]->mol!=NULL)
                continue;
        Molecule *m=new Molecule();
        m->addAtomRec(parent->atoms[i]);
    }
}

bool MolecList::verify() {
    bool tests[m];
    #pragma omp parallel for
    for (int i=0; i<m; ++i) {
        if (list[i]==NULL) {
                tests[i]=1;
                continue;
        } else {
                tests[i]=list[i]->verify();
        }
    }
    bool r=1;
    for (int i=0; i<m; ++i) {
        if (!tests[i]) {
                cout << "Molecule " << i << " gots a prob" << endl;
                r=0;
        }
    }
    return r;
}

void MolecList::combine(vector<Molecule*> mols) {
    Molecule* newM=new Molecule();
    cout << c << ' ' << m << endl;
    addElem(newM);
    cout << c << ' ' << m << endl;
    for (int i=0; i<mols.size(); ++i) {
        //DEB(Transfering atoms)
        mols[i]->transfer(newM);
        //DEB(Removing Molecule)
        mols[i]->parent->remElem(mols[i]->lIndex);
    }
    cout << c << ' ' << m << endl;
    cout << "molecules combined, adding this new one and compressing" << endl;
    compress();
    cout << c << ' ' << m << endl;
}

void MolecList::split(int tbs, vector<Molecule*> &out) {
    Molecule* splitter=list[tbs];
    while(splitter->atoms.size()>0) {
        Molecule* newm = new Molecule();
        newm->addAtom(*(splitter->atoms.begin()));
        vector<Atom*> adds;
        do {
                adds.clear();
                for (std::list<Atom*>::iterator i=newm->atoms.begin(); i!=newm->atoms.end(); ++i)
    {
                        vector<Atom*> bonded;
                        (*i)->getBonded(bonded);
                        for (int j=0; j<bonded.size(); ++j) {
                                if (bonded[j]->mol!=splitter)
                                        continue;
                                if (find(newm->atoms.begin(),newm->atoms.end(),bonded[j])==newm-
```

```
    >atoms.end())
                                        adds.push_back(bonded[j]);
                        }
                }
                for (int i=0; i<adds.size(); ++i) {
                        newm->addAtom(adds[i]);
                }
        } while (adds.size()>0);
        addElem(newm);
    }
}

void MolecList::transAll(Particle &P, map<unsigned int,Atom*> &ax) {
    Molecule *mol,*nMol;
    for (int i=0; i<P.molecules.max(); ++i) {
        if (P.molecules[i]==NULL) continue;
        mol=P.molecules[i];
        nMol=new Molecule();
        for (std::list<Atom*>::iterator j=mol->atoms.begin(); j!=mol->atoms.end(); ++j) {
                nMol->addAtom(ax[(*j)->lIndex]);
        }
        addElem(nMol);
    }
}
```

B.4 Element Type Classes

Type classes contain information including mass and force field parameters for the various types of atoms, bonds, angles, etc…

Atom Type Declaration — **atomT.h**

```
#ifndef ATOMT_H_
#define ATOMT_H_

class atomT
{
public:
    atomT() {}
    atomT(FLOAT m) {mass=m;}
    virtual ~atomT() {}

    FLOAT mass, eps, sigma;
};

#endif /*ATOMT_H_*/
```

Bond Type Declaration — **bondT.h**

```
#ifndef BONDT_H_
#define BONDT_H_

class bondT
{
public:
    bondT() {}
    bondT(FLOAT p1, FLOAT p2) { K=p1; r0=p2; }
    virtual ~bondT() {}
```

```
    FLOAT K, r0;
};

#endif /*BONDT_H_*/
```

Angle Type Declaration **angleT.h**

```
#ifndef ANGLET_H_
#define ANGLET_H_

class angleT
{
public:
    angleT() {}
    angleT(FLOAT p1, FLOAT p2) { K=p1; th0=p2;}
    virtual ~angleT() {}

    FLOAT K, th0;
};

#endif /*ANGLET_H_*/
```

Dihedral Type Declaration **dihedT.h**

```
#ifndef DIHEDT_H_
#define DIHEDT_H_

class dihedT
{
public:
    dihedT();
    dihedT(FLOAT P[]);
    virtual ~dihedT();

    string* type;

    void setType(char t[]);

    FLOAT p[4];
};

#endif /*DIHEDT_H_*/
```

Dihedral Type Function Definitions **dihedT.cpp**

```
#include "inc.h"
#include "dihedT.h"

dihedT::dihedT()
{
    type = NULL;
}

dihedT::~dihedT()
{
    if (type!=NULL)                                    //delete the type string
        delete type;
}

dihedT::dihedT(FLOAT P[]) {
```

```
    type = NULL;
    for (int i=0; i<4; i++)
        p[i]=P[i];
}

void dihedT::setType(char t[]) {
    type = new string(t);
}
```

Improper Type Declaration — **impT.h**

```
#ifndef IMPT_H_
#define IMPT_H_

class impT
{
public:
    impT() {}
    impT(FLOAT k, FLOAT X0) { K=k; x0=X0; }
    virtual ~impT() {}

    FLOAT K, x0;
};

#endif /*IMPT_H_*/
```

B.5 Element Type List Classes

Type lists serve as aggregators for the typing information. Indexing and memory functions are derived from the parent template class, so the specific classes handle data input and output as well as type transfer when copying between different particle classes. Like the element lists, all the type lists bare similarity in terms of memory handling and indexing, though because these lists must be held more static, and are generally shorter, they are derived from a different 'TypeList' template class.

TypeList.h

```
#ifndef TYPELIST_H_
#define TYPELIST_H_

class Particle;

template <typename T>
class TypeList
{
public:`
    TypeList();
    virtual ~TypeList();
    void addType(T* in,int index); //Adds type to the list, with an integer to be used as the
    calling index
    void remType(T*);           //Removes type by pointer (slow)
```

```
    void remType(int);          //Removes type by index
    void reserve(int k=5);      //Reserves space for types
    int size();                 //returns number of types stored in the list

    map<int,T*> typeInd;        //maps integer indices to types
    int m,c;                    //m=allocated memory, c=# of stored types
    T** types;          //stored types, dynamically allocated array
    Particle* parent;           //pointer to particle container
    T* operator[](int);         //recast of the array operator, given the type index, returns a
    pointer to the type
    int operator[](T*);         //recast of array operator, given the pointer, returns the type
    index
    int maxType();              //returns the highest index
};

template <typename T>
void TypeList<T>::reserve(int k) {
    if (k==0 && m==1) {   //in this case, the type list was merely constructed, backdoor method
    for deallocating
        free(types);
        return;
    } else if (k==0) return;     //k=0 means doesn't reserve anything
    if (m==1 && c==0) {          //in the event that m=1, this was only constructed, so k can be
    used to allocate the foreseeable required size
        types=(T**)realloc(types,k*sizeof(T*));
        m=k;
        for (int i=1; i<m; i++)
                types[i]=NULL;   //set new pointers to null
    } else {                     //this case, types had already been allocated, so allocate k
    more
        int ns=m+k;
        types=(T**)realloc(types,ns*sizeof(T*));
        for (int i=m; i<ns; i++) //set new pointers to null
                types[i]=NULL;
        m=ns;
    }
}

template <typename T>
TypeList<T>::TypeList<T>() { //default constructor allocates just one type to null
    m=1; c=0;
    types=(T**)malloc(sizeof(T*));
    types[0]=NULL;
}

template <typename T> //deallocates types themselves and the container storing them
TypeList<T>::~TypeList<T>() {
    if (c==0) { //no types are stored, just need to deallocate the container
        free(types);
        return;
    }
    for (int i=0; i<m; ++i) { //delete each type, types are allocated wiht new and therefore
    deallocated with delete
        if (types[i]!=NULL) {
                delete types[i];
                --c;
        }
    }
    if (c!=0)  //just a check to see if hte counter worked properly
        cerr << "Apparently not all types removed" << endl;
    free(types);
}

template <typename T>
void TypeList<T>::addType(T* n, int i) {
```

```
    if (m==c) //if container is full, add more capacity
        reserve();
    for (int j=0; j<m; ++j) {
        if (types[j]==NULL) {  //searching for a null pointer that can be used
                types[j]=n;  //stores the type
                typeInd[i]=n; //maps the type to the given index
                ++c;
                return;
        }
    }
}

template <typename T>
void TypeList<T>::remType(T* t) {
    for (int i=0; i<m; ++i) {
        if (types[i]==t) { //search the container for a poiner to the same location
                for (typename map<int,T*>::iterator j=typeInd.begin(); j!=typeInd.end(); ++j) {
                        if (j->second==t) {
                                typeInd.erase(j);  //deindex
                                break;
                        }
                }
                delete types[i]; //deallocate the type
                types[i]=NULL;   //set pointer to null so it can be used again
                --c;
        }
    }
}

template <typename T>
int TypeList<T>::operator[](T* k) {
    for (typename map<int,T*>::iterator j=typeInd.begin(); j!=typeInd.end(); ++j) {
        if (j->second==k) //search for hte matching pointer, return the index
                return j->first;
    }
    cerr << "Can't find atom type" << endl;
    return 0;
}

template <typename T>
void TypeList<T>::remType(int t) {
    typename map<int,T*>::iterator e=typeInd.find(t);
    for (int i=0; i<m; ++i) {
        if (types[i]==e->second) { //search for the type with the matching index
                delete types[i]; //deallocate type
                types[i]=NULL;
                --c;
                break;
        }
    }
    typeInd.erase(e);
}

template <typename T>
int TypeList<T>::size() {
    return c;
}

template <typename T>
T* TypeList<T>::operator[](int k) {
    return typeInd[k];
}

template <typename T>
int TypeList<T>::maxType() {
```

```
    return typeInd.rbegin()->first;
}


#endif /*TYPELIST_H_*/
```

Atom Type List Declaration **AtomTypeList.h**

```
#ifndef ATOMTYPELIST_H_
#define ATOMTYPELIST_H_

#include "restReader.h"
#include "data.h"

class AtomTypeList : protected TypeList<atomT>
{
public:
    AtomTypeList();
    virtual ~AtomTypeList();

    using TypeList<atomT>::parent;
    using TypeList<atomT>::size;
    using TypeList<atomT>::operator[];
    using TypeList<atomT>::maxType;
    using TypeList<atomT>::addType;

    void readMass(istream&, int);
    void readPairs(istream&);
    void readATypes(Data &data);
    void dataOut(ostream&);
    void transfer(AtomTypeList &t);

    void addDummyType();

    void harden();
    void soften();
    FLOAT*** prepPairCoefs();
    void freeCoeffs(FLOAT*** pairCoeffs);
};

#endif /*ATOMTYPELIST_H_*/
```

Atom Type List Definitions **AtomTypeList.cpp**

```
#include "inc.h"

AtomTypeList::AtomTypeList()
{
}

AtomTypeList::~AtomTypeList()
{
}

void AtomTypeList::readMass(istream& in, int k) {
    reserve(k);
    for (int i=0; i<k; ++i) {
        FLOAT mass; int index;
        in >> index >> mass;
        if (mass==100000)
                continue;
        atomT* at=new atomT(mass);
        addType(at,index);
```

```
    }
}

void AtomTypeList::readPairs(istream& in) {
    for (int i=0; i<c; ++i) {
        if (in.eof()) return;
        FLOAT eps, sigma; int index;
        if (!parent->soft)
                in >> index >> eps >> sigma;
        else
                in >> index >> eps;
        if (typeInd.find(index)==typeInd.end()) {
                i--;
                continue;
        }
        typeInd[index]->eps=eps;
        typeInd[index]->sigma=sigma;
    }
}

void AtomTypeList::readATypes(Data &data) {
    atomT* tmp;
    for (int i=1; i<=data.ntypes; ++i) {
        addType(tmp = new atomT(data.mass[i]),i);
        tmp->eps = data.pair_lj_epsilon[i];
        tmp->sigma = data.pair_lj_sigma[i];
    }
}


void AtomTypeList::dataOut(ostream& out) {
    out << "Masses" << endl << endl;
    for (int i=1; i<=maxType(); ++i) {
        if (typeInd.find(i)!=typeInd.end()) {
                atomT* t=typeInd[i];
                out << setw(3) << left << i << t->mass << endl;
        } else {
                out << setw(3) << left << i << 100000 << endl;
        }
    }
    if (parent->molecular) {
        out << endl << "Pair Coeffs" << endl << endl;
        for (int i=1; i<=maxType(); ++i) {
                if (typeInd.find(i)!=typeInd.end()) {
                        atomT* t=typeInd[i];
                        if (parent->soft)
                                out << setw(4) << left << i << setw(6) << setprecision(4) <<
    left << 3.0 << setprecision(4) << ' ' << 3.0 << endl;
                        else
                                out << setw(4) << left << i << setw(6) << setprecision(4) <<
    left << t->eps << setprecision(4) << ' ' << t->sigma << endl;
                } else {
                        out << setw(3) << left << i << setw(6) << left << 0.0 << setw(6) << left
    << 0.0 << endl;
                }
        }
    }
    out << endl;
}

void AtomTypeList::transfer(AtomTypeList &t) {
    for (map<int,atomT*>::iterator i=t.typeInd.begin(); i!=t.typeInd.end(); ++i) {
        if (typeInd.find(i->first)!=typeInd.end()) continue;
        atomT* old=i->second;
        int ind=i->first;
```

```
        atomT* a=new atomT(old->mass);
        a->eps=old->eps;
        a->sigma=old->sigma;
        addType(a,ind);
    }
}

void AtomTypeList::soften() {
    for (int i=0; i<m; ++i) {
        if (types[i]==NULL)
                continue;
        types[i]->eps=4;
        types[i]->sigma=10;
    }
}

void AtomTypeList::harden() {
    for (map<int, atomT*>::iterator i=typeInd.begin(); i!=typeInd.end(); ++i) {
        switch (i->first) {
                case 1:
                case 3:
                case 5:
                case 8:
                        i->second->eps=0.1;
                        i->second->sigma=4.0;
                        break;
                case 2:
                case 4:
                case 10:
                        i->second->eps=0.17;
                        i->second->sigma=3.0;
                        break;
                case 6:
                        i->second->eps=0.17;
                        i->second->sigma=3.12;
                        break;
                case 7:
                        i->second->eps=0.0;
                        i->second->sigma=0.0;
                        break;
                case 9:
                case 11:
                case 20:
                        i->second->eps=0.066;
                        i->second->sigma=3.5;
                        break;
                case 12:
                case 21:
                        i->second->eps=0.03;
                        i->second->sigma=2.5;
                        break;
        }
    }
}

void AtomTypeList::addDummyType() {
    atomT *nt;
    addType(nt = new atomT(100.0),maxType()+1);
    nt->eps = 0;
    nt->sigma = 0;
}
```

Bond Type List Declaration **BondTypeList.h**

```
#ifndef BONDTYPELIST_H_
#define BONDTYPELIST_H_

#include "restReader.h"
#include "data.h"

class BondTypeList : protected TypeList<bondT>
{
public:
    BondTypeList();
    virtual ~BondTypeList();

    using TypeList<bondT>::parent;
    using TypeList<bondT>::size;
    using TypeList<bondT>::operator[];
    using TypeList<bondT>::maxType;
    using TypeList<bondT>::addType;

    void readBTypes(istream&, int);
    void readBTypes(Data &data);
    void writeBTypes(ostream&);
    void transfer(BondTypeList &t);
};

#endif /*BONDTYPELIST_H_*/
```

Bond Type List Definitions **BondTypeList.cpp**

```
#include "inc.h"
#include "bondT.h"
#include "BondTypeList.h"

BondTypeList::BondTypeList()
{
}

BondTypeList::~BondTypeList()
{
}

void BondTypeList::readBTypes(istream& in, int k) {
    reserve(k);
    for (int i=0; i<k; ++i) {
        FLOAT K, rn; int index;
        in >> index >> K >> rn;
        if (K==0 && rn==0)
                continue;
        bondT* nb=new bondT(K,rn);
        addType(nb,index);
    }
}

void BondTypeList::readBTypes(Data &data) {
    for (int i=1; i<=data.nbondtypes; ++i) {
        addType(new bondT(
                data.bond_harmonic_k[i],data.bond_harmonic_r0[i])
                ,i);
    }
}

void BondTypeList::writeBTypes(ostream& out) {
    if (c==0)
        return;
    out << "Bond Coeffs" << endl << endl;
```

```
    for (int i=1; i<=maxType(); ++i) {
        if (typeInd.find(i)!=typeInd.end()) {
                bondT* t=typeInd[i];
                out << setw(3) << left << i << setw(10) << setprecision(4) << left << t->K <<
    setprecision(4) << t->r0 << endl;
        } else {
                out << setw(3) << left << i << setw(5) << left << 0.0 << setw(5) << left << 0.0
    << endl;
        }
    }
    out << endl;
}

void BondTypeList::transfer(BondTypeList &t) {
    for (map<int,bondT*>::iterator i=t.typeInd.begin(); i!=t.typeInd.end(); ++i) {
        if (typeInd.find(i->first)!=typeInd.end()) continue;
        bondT* b=new bondT(i->second->K,i->second->r0);
        addType(b,i->first);
    }
}
```

Angle Type List Declaration **AngleTypeList.h**

```
#ifndef ANGLETYPELIST_H_
#define ANGLETYPELIST_H_


#include "restReader.h"
#include "data.h"

class AngleTypeList : protected TypeList<angleT>
{
public:
    AngleTypeList();
    virtual ~AngleTypeList();

    using TypeList<angleT>::parent;
    using TypeList<angleT>::size;
    using TypeList<angleT>::operator[];
    using TypeList<angleT>::maxType;
    using TypeList<angleT>::addType;

    void readAngTypes(istream&, int);
    void readAngTypes(Data &data);
    void writeAngTypes(ostream&);
    void transfer(AngleTypeList &t);
};

#endif /*ANGLETYPELIST_H_*/
```

Angle Type List Definitions **AngleTypeList.cpp**

```
#include "inc.h"

AngleTypeList::AngleTypeList()
{
}

AngleTypeList::~AngleTypeList()
{
}
```

```
void AngleTypeList::readAngTypes(istream& in, int k) {
    reserve(k);
    for (int i=0; i<k; ++i) {
        FLOAT k, thn; int index;
        in >> index >> k >> thn;
        if (k==0.0 && thn==0.0)
                continue;
        angleT* t=new angleT(k,thn);
        addType(t,index);
    }
}

void AngleTypeList::readAngTypes(Data &data) {
    for (int i=1; i<=data.nangletypes; ++i) {
        addType(new angleT(data.angle_harmonic_k[i],data.angle_harmonic_theta0[i]*R2D),i);
    }
}

void AngleTypeList::writeAngTypes(ostream& out) {
    if (c==0)
        return;
    out << "Angle Coeffs" << endl << endl;
    for (int i=1; i<=maxType(); ++i) {
        if (typeInd.find(i)!=typeInd.end()) {
                angleT* t=typeInd[i];
                out << setw(3) << left << i << setw(10) << setprecision(4) << left << t->K <<
    setprecision(4) << t->th0 << endl;
        } else {
                out << setw(3) << left << i << setw(5) << left << 0.0 << setw(5) << left << 0.0
    << endl;
        }
    }
    out << endl;
}

void AngleTypeList::transfer(AngleTypeList &t) {
    for (map<int,angleT*>::iterator i=t.typeInd.begin(); i!=t.typeInd.end(); ++i) {
        if (typeInd.find(i->first)!=typeInd.end()) continue;
        angleT* a=new angleT(i->second->K,i->second->th0);
        addType(a,i->first);
    }
}
```

Dihedral Type List Declaration — DihTypeList.h

```
#ifndef DIHTYPELIST_H_
#define DIHTYPELIST_H_

#include "restReader.h"
#include "data.h"

class DihTypeList : protected TypeList<dihedT>
{
public:
    DihTypeList();
    virtual ~DihTypeList();

    using TypeList<dihedT>::parent;
    using TypeList<dihedT>::size;
    using TypeList<dihedT>::operator[];
    using TypeList<dihedT>::maxType;

    void readDTypes(istream&, int);
    void readDTypes(Data &data);
```

```
    void writeDTypes(ostream&);
    void transfer(DihTypeList &t);
};

#endif /*DIHTYPELIST_H_*/
```

Dihedral Type List Definition **DihTypeList.cpp**

```
#include "inc.h"
#include "dihedT.h"
#include "DihTypeList.h"

DihTypeList::DihTypeList()
{
}

DihTypeList::~DihTypeList()
{
}

void DihTypeList::readDTypes(istream& in, int k) {
    reserve(k);
    for (int i=0; i<k; ++i) {
        FLOAT p[4]; int index;
        in >> index >> p[0] >> p[1] >> p[2] >> p[3];
        if (p[0]==0 && p[1]==0 && p[2]==0 && p[3]==0)
                continue;
        dihedT* t=new dihedT(p);
        addType(t,index);
    }
}

void DihTypeList::readDTypes(Data &data) {
    FLOAT params[4];
    for (int i=1; i<=data.ndihedraltypes; ++i) {
        params[0] = 2.0 * data.dihedral_opls_k1[i];
        params[1] = 2.0 * data.dihedral_opls_k2[i];
        params[2] = 2.0 * data.dihedral_opls_k3[i];
        params[3] = 2.0 * data.dihedral_opls_k4[i];
        addType(new dihedT(params)
                ,i);
    }
}

void DihTypeList::writeDTypes(ostream& out) {
    if (c==0)
        return;
    out << "Dihedral Coeffs" << endl << endl;
    for (int i=1; i<=maxType(); ++i) {
        if (typeInd.find(i)!=typeInd.end()) {
                dihedT* t=typeInd[i];
                out << setw(3) << left << i;
                for (int k=0; k<4; k++)
                        out << setw(8) << setprecision(4) << left << t->p[k];
                out << endl;
        } else {
                out << setw(3) << left << i << setw(5) << left << 0.0 << setw(5) << left << 0.0;
                out << setw(5) << left << 0.0 << setw(5) << left << 0.0 << endl;
        }
    }
    out << endl;
}

void DihTypeList::transfer(DihTypeList &t) {
```

```
    for (map<int,dihedT*>::iterator i=t.typeInd.begin(); i!=t.typeInd.end(); ++i) {
        if (typeInd.find(i->first)!=typeInd.end()) continue;
        dihedT* d=new dihedT(i->second->p);
        addType(d,i->first);
    }
}
```

Improper Type List Declaration — **ImpTypeList.h**

```
#ifndef IMPTYPELIST_H_
#define IMPTYPELIST_H_

#include "restReader.h"
#include "data.h"

class ImpTypeList : protected TypeList<impT>
{
public:
    ImpTypeList();
    virtual ~ImpTypeList();

    using TypeList<impT>::parent;
    using TypeList<impT>::size;
    using TypeList<impT>::operator[];
    using TypeList<impT>::maxType;

    void readImpTypes(istream&, int);
    void readImpTypes(Data &data);
    void writeImpTypes(ostream&);
    void transfer(ImpTypeList &t);
};

#endif /*IMPTYPELIST_H_*/
```

Improper Type List Definitions — **ImpTypeList.cpp**

```
#include "inc.h"
#include "impT.h"
#include "ImpTypeList.h"

ImpTypeList::ImpTypeList()
{
}

ImpTypeList::~ImpTypeList()
{
}

void ImpTypeList::readImpTypes(istream& in, int k) {
    reserve(k);
    for (int i=0; i<k; ++i) {
        int index; FLOAT K, X0;
        in >> index >> K >> X0;
        if (K==0 && X0==0)
                continue;
        impT* t=new impT(K,X0);
        addType(t,index);
    }
}

void ImpTypeList::readImpTypes(Data &data) {
    for (int i=1; i<=data.nimpropertypes; ++i) {
```

```
        addType(new impT(
                (FLOAT)data.improper_harmonic_k[i],
                (FLOAT)data.improper_harmonic_chi[i])
                ,i);
    }
}

void ImpTypeList::writeImpTypes(ostream& out) {
    if (c==0)
        return;
    out << "Improper Coeffs" << endl << endl;
    for (int i=1; i<=maxType(); ++i) {
        if (typeInd.find(i)!=typeInd.end()) {
                impT* t=typeInd[i];
                out << setw(3) << left << i << setw(10) << setprecision(4) << left << t->K <<
    setprecision(4) << t->x0 << endl;
        } else {
                out << setw(3) << left << i << setw(5) << left << 0.0 << setw(5) << left << 0.0
    << endl;
        }
    }
    out << endl;
}

void ImpTypeList::transfer(ImpTypeList &t) {
    for (map<int,impT*>::iterator i=t.typeInd.begin(); i!=t.typeInd.end(); ++i) {
        if (typeInd.find(i->first)!=typeInd.end()) continue;
        impT* imp=new impT(i->second->K,i->second->x0);
        addType(imp,i->first);
    }
}
```

B.6 Main Particle Class

The ‘particle’ class brings all the extraneous bits together, and operations are routed through this class.

Particle class declaration **particle.h**

```
class Particle
{
public:
    Particle();
    virtual ~Particle();

    //Element Type Lists
    AtomTypeList aTypes;
    BondTypeList bTypes;
    AngleTypeList angTypes;
    DihTypeList dTypes;
    ImpTypeList iTypes;

    //Element Lists
    AtomList atoms;
    BondList bonds;
    AngleList angles;
    DihedList dihedrals;
    ImpList impropers;
    MolecList molecules;
```

```
    //Dimensions and dimension handling functions
    FLOAT* dim;
    FLOAT* perDim;
    bool periodic[3];
    bool molecular;
    void setFlat();

    //Simulation Snapshot Handling
    int dumps;
    bigint* timesteps;
    void resForDump(int);
    void periodicCheck(FLOAT* in, int s, int d, int count);

    //Data and snapshot reading functions
    void readDataFull(istream&);
    void readDataFull(char fn[]);
    void readDataShort(istream&);                //Short reads take in only atomic positions and
    bonds, typically used for analysis since other topology information is not necessary
    void readDataShort(char fn[]);
    void readDataBin(char fn[], bool full);              //Bin reader takes in restart file,
    LAMMPS version is important for compatibility
    void scanData(istream&,unsigned int*);
    bool fileSeek(istream&,const string);
    int dumpSeek(istream&,const string);
    void readDumpFile(char*);
    void readDumpAscii(istream &in);
    void readDumpBin(istream &in);               //Reads binary dump files, much faster to read
    and faster writing during runs, binary dump is modified so headers come through

    //Data Output Functions
    void writeDataFile(ostream&,int k=0);
    void writeMol2File(ostream&,int k=0);
    void writeMol2AllDumps(ostream& out);
    void writeDumpsMol2(ostream&);
    void verifyCon(int k);
    bool soft;

    //Film Measurement Functions
    void calcAllDih(ostream&);          //Calculates C-C-C-C dihedral angles of alkylsilanes
    bound to the surface
    void calcHexDih(ostream&);          //Calculates C-C-C-C dihedral angles of hexane solvent
    molecules

    void compressAll();
    void checkPos();                            //makes sure all atoms are within the bounds of
    the box
    void resetConnectivities();

    //Functionalizing functions
    void functFlatUni(istream& molIn, int count);//Functionalizes flat surface with uniform
    distribution
    void functUni(istream& molIn, int count);           //Functionalizes particle surface with
    uniform distribution
    void readSilane(istream& in);                                       //Reads in the silane
    molecule and sets up bonding topology
    void flipXZ();                                                                  //Flips
    molecule for placement onto flat surface
    int remoteH(vector<Atom*> &hydrogens, vector<Atom*> &silanes);              //Calls
    specific remoteH functions
    int remoteHRound(vector<Atom*> &hydrogens, vector<Atom*> &silanes);  //Finds H with greatest
    distance to functionalized sites on particle surface, uses radial distance
    int remoteHFlat(vector<Atom*> &hydrogens, vector<Atom*> &silanes);   //Finds H with greatest
    distance to functionalized sites on flat surface, uses direct distance with PBC
    Atom* appendMol(Atom* functHere, Particle &molyToAppend);                   //Places
    molecule at given H
```

```
    void silaneBridge();
                        //Performs bridging of silanes
    void makeBridge(Atom* si1, Atom* si2, Molecule *bulk);                                  //Makes
    a bridge between specific silicon atoms of silane, bulk molecule given for O assignment
    void makeBridge(Atom* si1, Atom* O, Atom *si2);
        //Makes a bridge using a preexisting hydroxyl given by O, removes the H
    void addHydroxyls(Atom* sil, Molecule *bulk);                                           //Adds
    hydroxyls where necessary
    FLOAT top();
                                //Finds point at which top boundary can be set, highest atom in
    z direction
    void checkOs();
                                        //Checks hydroxyl and surface atom types



    //Parent function for effective surface coverage determination
    void calcCoverageProps(ostream& covOut, ostream& distOut, ostream& covMol2, ostream& dirOut,
    FLOAT radius, FLOAT angle, FLOAT meshRad=1);
    //Calculates coverage for round surface, returns surface mesh points
    list<Atom*> calcCovAllRound(ostream&, ostream&, FLOAT, FLOAT, FLOAT*& centers, FLOAT
    meshRad=1);
    //Same as above for flat surface
    list<Atom*> calcCovAllFlat(ostream&, FLOAT, FLOAT meshRad=1.0);
    //Prints the surface mesh with type indicators for exposed and covered sites
    void printCovMol2(ostream& mol2Out, FLOAT* points, bool* cov, int dumps, int points);
    //Calculates radial distribution of hydrocarbons on surface, a measure of film thickness,
    outputs histogram
    void calcRaDistRound(ostream&, FLOAT*,list<Atom*>&);
    void calcRaDistFlat(ostream&, list<Atom*>&);
    //Determines center of mass of the particle for placing spherical mesh
    FLOAT calcCenter(vector<Atom*>&, FLOAT* out, int s=0, int d=1);
    //Determines direction of C-C bonds relative to surface normal, outputs raw directions for
    later binning
    void calcDirections(ostream& out, FLOAT* centers=NULL);
    void getFlatDumpHeights(FLOAT*);
    //Atom parsing routines, named straightforwardly
    void getAtomsByType(list<Atom*>&, vector<atomT*>&);
    void getAtomsByType(vector<Atom*>&, vector<atomT*>&);
    void parseByDist(list<Atom*>&,vector<Atom*>&, FLOAT* point, FLOAT dist);
    void getSilAtoms(vector<Atom*>&);
    void checkCov(FLOAT* point, FLOAT* pointDir, vector<Atom*>& relAtoms, bool* flags, FLOAT cut,
    int s, int d);

    //periodic cell replication functions
    void replicate(int xr, int yr, int zr);
    void singDimRep(int dim, int n);

    void bound(FLOAT ang, FLOAT space);

    //Soften and harden flip forcefield between soft cosine and OPLS potentials
    void soften();
    void harden();

    //3-dimensional rotation based on three rotational angles
    void rotate(FLOAT,FLOAT,FLOAT);

    //removes a molecule from the structure, deleting atoms and topology
    void delMol(int);

    //Moves the whole system
    void translate(FLOAT t[]);
    //moves the whole system accomodating periodic boundaries
    void translatePer(FLOAT t[]);
    //Adds particle P to this particle
```

```
    void combineParticles(Particle &P);
    void transAll(Particle &P);
    //Removes particles beyond a cutoff 'cut' in dimension 'd'
    void cleave(FLOAT cut, int d);
    //Removes particles in region specified by 'c', see definition
    void cleave(FLOAT *c);
    //Removes atoms in a cylinder along dimension d wiht radius 'rMax'
    void cleaveCyl(FLOAT rMax, int d);
    //Make everything a single molecule
    void make1Mol();

    //Range-of-motion calculation
    void ROM(int interval, ostream &out);
    //PBC corrections to move atoms back into the cell
    void minimumImage(FLOAT &x, FLOAT &y, FLOAT &z, int k);
    void minimumImage(FLOAT *d, int k);
};
```

Particle Function Definitions **particle.cpp**

```
#include <omp.h>
#include <string>
#include "inc.h"


Particle::Particle()
{
    aTypes.parent=this;
    bTypes.parent=this;
    angTypes.parent=this;
    dTypes.parent=this;
    iTypes.parent=this;
    atoms.parent=this;
    bonds.parent=this;
    angles.parent=this;
    dihedrals.parent=this;
    impropers.parent=this;
    molecules.parent=this;
    molecular = true;
    periodic[0]=0; periodic[1]=0; periodic[2]=0;
    #ifdef ipp
        dim=IPPF1(ippsMalloc)(6);
        perDim=IPPF1(ippsMalloc)(3);
        cout << dim << ' ' << perDim << endl;
        cout << dim[0] << ' ' << perDim[0] << endl;
    #else
        dim=(FLOAT*)malloc(sizeof(FLOAT)*6);
        perDim=(FLOAT*)malloc(sizeof(FLOAT)*3);
    #endif
    dumps=0;
    timesteps=NULL; dumps=0;
    soft=0;
}

Particle::~Particle()
{
    molecules.clear();
    bonds.clear();
    angles.clear();
    dihedrals.clear();
    impropers.clear();
    atoms.clear();
    #ifdef ipp
        ippsFree(dim);
```

```
        ippsFree(perDim);
    #else
        free(dim);
        free(perDim);
    #endif
    if (dumps>0)
        free(timesteps);
}

void Particle::setFlat() {
    periodic[0]=1;
    periodic[1]=1;
}

void Particle::resForDump(int dc) {
    int ns;
    if (dumps==0) {
        ns=dc;
        timesteps=(bigint*)malloc(ns*sizeof(bigint));
    } else {
        ns=dumps+dc;
        timesteps=(bigint*)realloc(timesteps,ns*sizeof(bigint));
    }
    DEB(timesteps)
    #ifdef ipp
        //DEB(1)
        FLOAT* nDim=IPPF1(ippsMalloc)(6*(ns+1));
        IPPF1(ippsCopy)(dim,nDim,6*(dumps+1));
        //DEB(1.5)
        ippsFree(dim);
        //DEB(2)
        dim=nDim;
        FLOAT* nPerDim=IPPF1(ippsMalloc)(3*(ns+1));
        IPPF1(ippsCopy)(perDim,nPerDim,3*(dumps+1));
        ippsFree(perDim);
        //DEB(3)
        perDim=nPerDim;
    #else
        dim=(FLOAT*)realloc(dim,6*(ns+1)*sizeof(FLOAT));
        perDim=(FLOAT*)realloc(perDim,3*(ns+1)*sizeof(FLOAT));
    #endif
    dumps=ns;
}

void Particle::periodicCheck(FLOAT* in, int s, int d, int count) {
    int s3=3*s, d3=3*d;
    for (int i=0; i<3; i++) {
        if (periodic[i]) {
                #pragma omp parallel for
                for (int j=0; j<d; ++j) {
                        int j3=3*j;
                        for (int k=0; k<count; ++k) {
                                if(fabs(in[j3+d3*k+i])>perDim[s3+j3+i]/2) {
                                        if (in[j3+d3*k+i]>0)
                                                in[j3+d3*k+i]-=perDim[s3+j3+i];
                                        else
                                                in[j3+d3*k+i]+=perDim[s3+j3+i];
                                }
                        }
                }
        }
    }
}

void Particle::verifyCon(int k) {
```

```
    molecules.verify();
    bonds.verify();
    angles.verify(k);
    dihedrals.verify(k);
    impropers.verify(k);
}

void Particle::soften() {
    aTypes.soften();
}

void Particle::harden() {
    aTypes.harden();
    soft=0;
}

void Particle::compressAll() {
    molecules.compress();
    atoms.compress();
    bonds.compress();
    angles.compress();
    dihedrals.compress();
    impropers.compress();
}

void Particle::bound(FLOAT ang, FLOAT space) {
    for (int i=0; i<molecules.size(); ++i) {
        if (molecules[i]==NULL)
                continue;
        else
                molecules[i]->bound(ang,space);
    }
}

void Particle::combineSolv() {
    vector<Molecule*> solvMols;
    for (int i=0; i<molecules.max(); ++i) {
        if (molecules[i]==NULL) continue;
        if (aTypes[(*molecules[i]->atoms.begin())->type]>=20)
                solvMols.push_back(molecules[i]);
    }
    if (solvMols.size()>1) {
        cout << "Combining " << solvMols.size() << " solvent molecules into one" << endl;
        molecules.combine(solvMols);
    }
}

void Particle::make1Mol() {
    vector<Molecule*> ms;
    for (int i=0; i<molecules.max(); ++i) {
        if (molecules[i]==NULL) continue;
        ms.push_back(molecules[i]);
    }
    if (ms.size()>1) {
        molecules.combine(ms);
    }
}

void Particle::delMol(int i) {
    molecules[i]->clearAtoms();
    molecules.remElem(i);
}

void Particle::resetConnectivities() {
    atoms.clearConnectivities();
```

```
    bonds.resetConn();
    angles.resetConn();
    dihedrals.resetConn();
    impropers.resetConn();
}

void Particle::minimumImage(FLOAT &x, FLOAT &y, FLOAT &z, int k) {
  FLOAT d[3]={x,y,z};
  minimumImage(d,k);
  x = d[0]; y = d[1]; z = d[2];
}

void Particle::minimumImage(FLOAT *d, int k) {
  if (k<0) k=dumps+1+k;
  FLOAT *pd = perDim+3*k;
  for (int i = 0; i < 3; ++i ) {
    if (periodic[i]) {
      while (d[i]>(pd[i]/2.0))
        d[i]-=pd[i];
      while (d[i]<(-pd[i]/2.0))
        d[i]+=pd[i];
    }
  }
}

void Particle::calcAllDih(ostream& out) {
    molecules.compress();
    int l[molecules.max()];
    FLOAT* diheds[molecules.max()];
    #pragma omp parallel for
    for (unsigned int i=0; i<molecules.max(); ++i) {
        if (molecules[i]==NULL) { l[i]=0; continue; }
        l[i]=molecules[i]->calcDihedChain(diheds[i],1,dumps);
    }
    for (int i=0; i<dumps; ++i) {
        out << "TIMESTEP: " << timesteps[i] << endl;
        unsigned int k=1;
        for (unsigned int j=0; j<molecules.max(); ++j) {
                if (l[j]==0) continue;
                out << setw(8) << left << k;
                k++;
                for (int f=0; f<l[j]; ++f) {
                        out << setw(15) << setprecision(10) << left <<  diheds[j][f*dumps+i];
                }
                out << endl;
        }
    }
    for (unsigned int i=0; i<molecules.max(); ++i) {
        #ifdef ipp
                if (l[i]!=0) ippsFree(diheds[i]);
        #else
                if (l[i]!=0) free(diheds[i]);
        #endif
    }
}
```

Particle Data Reading Functions **ParticleReadData.cpp**

```
#include <omp.h>
#include <string>
#include "inc.h"
#include <string.h>
#include <stdio.h>
#include "restReader.h"
```

```
#include "data.h"

void Particle::scanData(istream& in, unsigned int c[]) {
    //Initialization stuff, variable collection
    //Counts = {atoms, bonds, angles, dihedrals, types listed in same manner}
    char buf[120]; string lineStr;
    in.getline(buf,120);
    in.getline(buf,120);
    istringstream iss;
    in.seekg(ios::beg);
    //Collecting basic run parameters
    while (!in.eof() && lineStr.find("Masses")!=0) {
        getline(in,lineStr);
        if (lineStr.find("atoms")<10) {
                iss.str(lineStr);
                iss >> c[0];
        } else if (lineStr.find("bonds")<10) {
                iss.str(lineStr);
                iss >> c[1];
        } else if (lineStr.find("angles")<10) {
                iss.str(lineStr);
                iss >> c[2];
        } else if (lineStr.find("dihedrals")<10) {
                iss.str(lineStr);
                iss >> c[3];
        } else if (lineStr.find("impropers")<10) {
                iss.str(lineStr);
                iss >> c[4];
        } else if (lineStr.find("atom types")<10) {
                iss.str(lineStr);
                iss >> c[5];
        } else if (lineStr.find("bond types")<10) {
                iss.str(lineStr);
                iss >> c[6];
        } else if (lineStr.find("angle types")<10) {
                iss.str(lineStr);
                iss >> c[7];
        } else if (lineStr.find("dihedral types")<10) {
                iss.str(lineStr);
                iss >> c[8];
        } else if (lineStr.find("improper types")<20) {
                iss.str(lineStr);
                iss >> c[9];
        } else if (lineStr.find("xlo")<100) {
                iss.str(lineStr);
                iss >> dim[0] >> dim[1];
                perDim[0]=dim[1]-dim[0];
        } else if (lineStr.find("ylo")<100) {
                iss.str(lineStr);
                iss >> dim[2] >> dim[3];
                perDim[1]=dim[3]-dim[2];
        } else if (lineStr.find("zlo")<100) {
                iss.str(lineStr);
                iss >> dim[4] >> dim[5];
                perDim[2]=dim[5]-dim[4];
        }
    }
}

bool Particle::fileSeek(istream& in,const string s) {
    string lineStr;
    while (lineStr.find(s)==string::npos) {
        getline(in,lineStr);
        if (in.bad()) {
                in.clear();
```

```
                in.seekg(0,ios::beg);
        }
        if (in.eof()) {
                in.clear();
                in.seekg(0,ios::beg);
                return 0;
        }
    }
    //DEB(fseek)
    getline(in,lineStr);
    return 1;
}

int Particle::dumpSeek(istream& in, const string s) {
    string lineStr;
    while (lineStr.find(s)==string::npos) {
        getline(in,lineStr);
        if (in.eof()) {
                in.clear();
                in.seekg(ios::beg);
                return -1;
        }
    }
    //DEB(fseek)
    int tmp;
    in >> tmp;
    cout << tmp << endl;
    return tmp;
}

void Particle::readDataFull(char fn[]) {
    if (strstr(fn,".data")==NULL) {
        readDataBin(fn,1);
    } else {
        ifstream in(fn);
        if (!in.is_open()) {
                cout << "Invalid data file given" << endl;
                abort();
        }
        readDataFull(in);
        in.close();
    }
}

void Particle::readDataShort(char fn[]) {
    if (strstr(fn,".data")==NULL) {
        readDataBin(fn,0);
    } else {
        ifstream in(fn);
        if (!in.is_open()) {
                cout << "Invalid data file given" << endl;
                abort();
        }
        readDataShort(in);
        in.close();
    }
}

void Particle::readDataFull(istream& in) {
    DEB(Performing Full Read)
    unsigned int c[10]={0,0,0,0,0,0,0,0,0,0};
    DEB(Scanning Datafile) /* = cout << "Scanning Datafile" << endl; */
    scanData(in,c);
    in.seekg(0,ios::beg);
    DEB(Reading Type Information)
```

```
    if (fileSeek(in,"Masses"))
        aTypes.readMass(in,c[5]);
    if (fileSeek(in,"Pair Coeffs"))
        aTypes.readPairs(in);
    if (fileSeek(in,"Bond Coeffs"))
        bTypes.readBTypes(in,c[6]);
    if (fileSeek(in,"Angle Coeffs"))
        angTypes.readAngTypes(in,c[7]);
    if (fileSeek(in,"Dihedral Coeffs"))
        dTypes.readDTypes(in,c[8]);
    if (fileSeek(in,"Improper Coeffs"))
        iTypes.readImpTypes(in,c[9]);
    DEB(Reading Atoms)
    if (fileSeek(in,"Atoms")) {
        if (c[1]==0) {
                molecular = false;
                Molecule *nm = new Molecule;
                molecules.addElem(nm);
                atoms.readAtomsNoMol(in,c[0],nm);
        } else {
                atoms.readAtoms(in,c[0]);
        }
    }
    DEB(Reading Bonds)
    if (fileSeek(in,"Bonds"))
        bonds.readBonds(in,c[1]);
    DEB(Reading Angles)
    if (fileSeek(in,"Angles"))
        angles.readAngles(in,c[2]);
    DEB(Reading Dihedrals)
    if (fileSeek(in,"Dihedrals"))
        dihedrals.readDihedData(in,c[3]);
    DEB(Reading Impropers)
    if (fileSeek(in,"Impropers")) impropers.readImpData(in,c[4]);
    DEB(Done Reading Datafile)
}

void Particle::readDataShort(istream& in) {
    DEB(Performing Abbreviated Data Read)
    unsigned int c[10]={0,0,0,0,0,0,0,0,0,0};
    DEB(Scanning Data File)
    scanData(in,c);
    in.seekg(0,ios::beg);
    DEB(Reading Type Info)
    if (fileSeek(in,"Masses"))
        aTypes.readMass(in,c[5]);
    if (fileSeek(in,"Pair Coeffs"))
        aTypes.readPairs(in);
    if (fileSeek(in,"Bond Coeffs"))
        bTypes.readBTypes(in,c[6]);
    DEB(Reading Atoms)
    if (fileSeek(in,"Atoms"))
        atoms.readAtoms(in,c[0]);
    DEB(Reading Bonds)
    if (fileSeek(in,"Bonds"))
        bonds.readBonds(in,c[1]);
    DEB(Done Reading Datafile)
}

void Particle::readDumpFile(char fn[]) {
    if (strstr(fn,".bin")==NULL) {
        ifstream in(fn);
        if (!in.is_open()) {
                cout << "Invalid dump file given" << endl;
                abort();
```

```
        }
        readDumpAscii(in);
        in.close();
    } else {
        ifstream in(fn,ios::binary);
        if (!in.is_open()) {
                cout << "Invalid dump file given" << endl;
                abort();
        }
        readDumpBin(in);
        in.close();
    }
}

void Particle::readDumpBin(istream &in) {
    cout << "Performing binary dump file read" << endl;
    int dc = 0;
    bigint timestep,prev;
    int size_one,nchunk,n;
    double *buffer = new double[1000];
    int mbuf=1000;
    while (1) {
        in.read((char*)&timestep,sizeof(bigint));
        cout << timestep << " timestep found" << endl;
        if (in.eof()) break;
        in.read((char*)buffer,sizeof(bigint)+sizeof(int)*7+sizeof(double)*6);
        in.read((char*)&size_one,sizeof(int));
        in.read((char*)buffer,sizeof(int));
        in.read((char*)&nchunk,sizeof(int));
        for (int i=0; i<nchunk; ++i) {
                in.read((char*)&n,sizeof(int));

                if (n>mbuf) {
                        delete [] buffer;
                        buffer = new double[n];
                        mbuf = n;
                }
                in.read((char*)buffer,n*sizeof(double));
        }
        if (timestep!=prev) {
                dc++;
                prev=timestep;
        }
    }
    cout << dc << " dumps found" << endl;
    resForDump(dc);
    in.clear();
    in.seekg(0,ios::beg);
    int j, tmp, index;
    double *row;
    bool readOver=0;
    for (int i=0; i<dc; ++i) {
        j=i+1+dumps-dc;
        in.read((char*)&timestep,sizeof(bigint));
        if (in.eof()) break;
        if (timestep == timesteps[j-2] && j!=1) {
                in.read((char*)buffer,sizeof(bigint)+9*sizeof(int)+6*sizeof(double));
                in.read((char*)&nchunk,sizeof(int));
                for (int j=0; j<nchunk; ++j) {
                        in.read((char*)&n,sizeof(int));
                        in.read((char*)buffer,n*sizeof(double));
                }
                continue;
        }
        timesteps[j-1]=timestep;
```

```
        cout << "Reading timestep " << timesteps[j-1] << endl;
        //read box dimensions and store
        in.read((char*)buffer,sizeof(bigint)+sizeof(int)*7);
        in.read((char*)(dim+6*j),6*sizeof(double));
        for (int k=0; k<3; k++)
                perDim[3*j+k]=dim[6*j+2*k+1]-dim[6*j+2*k];
        //read each processor chunk
        in.read((char*)&size_one,sizeof(int));
        in.read((char*)buffer,sizeof(int));
        in.read((char*)&nchunk,sizeof(int));
        for (int k = 0; k < nchunk; ++k) {
                in.read((char*)&n,sizeof(int));
                in.read((char*)buffer,n*sizeof(double));
                n/=size_one;
                //#pragma omp parallel for
                for (int kk=0; kk < n; ++kk) {
                        row = buffer + size_one*kk;
                        index = (int)row[0];
                        if (i==0)
                                atoms.atmInd[index]->resDumps(dc);
                        for (int l=0; l<3; ++l)
                                row[2+l]=row[2 + l] * perDim[3*j + l] + dim[6*j + 2*l];
                        atoms.atmInd[index]->setPos(row + 2,j);
                }
        }
    }
    delete [] buffer;
}

void Particle::readDumpAscii(istream &in) {
    DEB(Reading Ascii Dump File)
    //stringstream in;
    //in << fileIn.rdbuf();
    string lineStr;
    //cout << "Fname" << fn << endl;
    int dc=0;
    int prev=-10, nw;
    while ((nw=dumpSeek(in,"TIMESTEP"))>=0) {
        if (nw!=prev) {
                prev=nw;
                dc++;
        }
    }
    resForDump(dc);
    #ifdef ipp
        FLOAT* pos=IPPF1(ippsMalloc)(3);
    #else
        FLOAT pos[3];
    #endif
    int j;
    bigint tmp;
    for (int i=0; i<dc; ++i) {
        j=i+1+dumps-dc;
        bool found=0;
        while (!found) {
                getline(in,lineStr);
                if (lineStr.find("TIMESTEP")!=string::npos) {
                        in >> tmp;
                        if (tmp!=timesteps[j-2] || j==1)
                                found=1;
                }
        }
        timesteps[j-1]=tmp;
        cout << "Reading timestep " << timesteps[j-1] << endl;
        while (lineStr.find("ITEM: NUMBER OF ATOMS")==string::npos)
```

```
                getline(in,lineStr);
        int acount;
        in >> acount;
        cout << "Reading " << acount << " atoms" << endl;
        while (lineStr.find("BOX BOUNDS")==string::npos) {
                //DEB(BOX1)
                getline(in,lineStr);
        }
        in >> dim[6*j] >> dim[6*j+1] >> dim[6*j+2];
        in >> dim[6*j+3] >> dim[6*j+4] >> dim[6*j+5];
        for (int k=0; k<3; k++)
                perDim[3*j+k]=dim[6*j+2*k+1]-dim[6*j+2*k];
        //DEB(ATOMS1)

        while (lineStr.find("ITEM: ATOMS")==string::npos)
                getline(in,lineStr);
        for (int k=0; k<acount; ++k) {
                unsigned int index;
                in      >> index;
                //cout << "Atom " << index << endl;
                int trash;
                in >> trash >> pos[0] >> pos[1] >> pos[2];
                for (int l=0; l<3; ++l)
                        pos[l]=pos[l]*perDim[3*j+l]+dim[6*j+2*l];
                if (i==0) {
                        atoms.atmInd[index]->resDumps(dc);
                }
                atoms.atmInd[index]->setPos(pos,j);
        }
    }
    #ifdef ipp
        ippsFree(pos);
    #endif
}

void Particle::readDataBin(char fn[],bool full=1) {
    FILE *fp = fopen(fn,"rb");
    if (fp == NULL) {
      printf("ERROR: Cannot open restart file %s\n",fn);
      abort();
    }
    Data data;

    header(fp,data);
    if (data.size_smallint != sizeof(int) ||
      data.size_tagint != sizeof(tagint) ||
      data.size_bigint != sizeof(bigint)) {
    printf("ERROR: Data type sizes in restart file "
                "are incompatible with restart2data.cpp\n");
    abort();
    }
    groups(fp);
    type_arrays(fp,data);
    force_fields(fp,data);
    modify(fp);
    double *buf = NULL;
    int n,m;
    int maxbuf = 0;
    data.iatoms = data.ibonds = data.iangles =
    data.idihedrals = data.iimpropers = 0;
    for (int iproc = 0; iproc < data.nprocs; iproc++) {
        n = read_int(fp);

        if (n > maxbuf) {
          maxbuf = n;
```

```
          delete [] buf;
          buf = new double[maxbuf];
        }

        nread_double(buf,n,fp);

        m = 0;
        while (m < n) m += atom(&buf[m],data);
    }
    fclose(fp);
    dim[0] = data.xlo;
    dim[1] = data.xhi;
    dim[2] = data.ylo;
    dim[3] = data.yhi;
    dim[4] = data.zlo;
    dim[5] = data.zhi;
    perDim[0] = dim[1] - dim[0];
    perDim[1] = dim[3] - dim[2];
    perDim[2] = dim[5] - dim[4];

    aTypes.readATypes(data);
    atoms.readAtoms(data);
#pragma omp parallel sections
    {
#pragma omp section
        {
                if (full && data.nbonds) {
                        bTypes.readBTypes(data);
                        bonds.readBonds(data);
                }
        }
#pragma omp section
        {
                if (full && data.nangles) {
                        angTypes.readAngTypes(data);
                        angles.readAngles(data);
                }
        }
#pragma omp section
        {
                if (full && data.ndihedrals) {
                        dTypes.readDTypes(data);
                        dihedrals.readDihedData(data);
                }
        }
#pragma omp section
        {
                if (full && data.nimpropers) {
                        iTypes.readImpTypes(data);
                        impropers.readImpData(data);
                }
        }
    }
}
```

Particle Writing Data Functions **ParticleWriteData.cpp**

```
#include <omp.h>
#include <string>
#include "inc.h"

void Particle::writeDataFile(ostream& out, int k) {
    out << "Lammps Datafile written by program coded by Brad Ewers" << endl << endl;
```

```
    out << atoms.size() << " atoms" << endl;
    if (bonds.size())
        out << bonds.size() << " bonds" << endl;
    if (angles.size())
        out << angles.size() << " angles" << endl;
    if (dihedrals.size())
        out << dihedrals.size() << " dihedrals" << endl;
    if (impropers.size())
        out << impropers.size() << " impropers" << endl;
    out << aTypes.maxType() << " atom types" << endl;
    if (bTypes.size())
        out << bTypes.maxType() << " bond types" << endl;
    if (angTypes.size())
        out << angTypes.maxType() << " angle types" << endl;
    if (dTypes.size())
        out << dTypes.maxType() << " dihedral types" << endl;
    if (iTypes.size())
        out << iTypes.maxType() << " improper types" << endl;
    out << endl;
    out << setw(15) << setprecision(10) << left << dim[0] << setw(15) << setprecision(10) << left
    << dim[1] << " xlo xhi" << endl;
    out << setw(15) << setprecision(10) << left << dim[2] << setw(15) << setprecision(10) << left
    << dim[3] << " ylo yhi" << endl;
    out << setw(15) << setprecision(10) << left << dim[4] << setw(15) << setprecision(10) << left
    << dim[5] << " zlo zhi" << endl;
    out << endl;
    aTypes.dataOut(out);
    bTypes.writeBTypes(out);
    angTypes.writeAngTypes(out);
    dTypes.writeDTypes(out);
    iTypes.writeImpTypes(out);
    atoms.writeAtomData(out,k);
    bonds.writeBondsData(out);
    angles.writeAngleData(out);
    if (dihedrals.size()>0)
        dihedrals.writeDihedData(out);
    if (impropers.size()>0)
        impropers.writeImpData(out);
}

void Particle::writeMol2File(ostream& out, int k) {
    elements el;
    out << "@<TRIPOS>MOLECULE" << endl;
    out << "Mol2 File prepared by system manager by Brad Ewers" << endl;
    out << atoms.dumped() << ' ' << bonds.nonBndCount(k) << endl;
    out << "SMALL" << endl << "GESTEIGER" << endl << "Energy=0" << endl << endl;
    atoms.writeAtomMol2(out,k);
    bonds.writeBondsMol2(out,k);
}

void Particle::writeMol2AllDumps(ostream& out) {
    for (int i=1; i<=dumps; ++i) {
        writeMol2File(out,i);
    }
}


void Particle::writeDumpsMol2(ostream& out) {
    for (int i=1; i<=dumps; ++i)
        writeMol2File(out,i);
}
```

Particle Functionalization Functions **ParticleFunct.cpp**

```
#include "inc.h"

void Particle::functUni(istream& molIn, int count) {
    Particle moly;
    moly.readSilane(molIn);
    prepCombine(moly);
    if (periodic[0]==1 && periodic[1]==1 && periodic[2]==0)
        moly.flipXZ();
    vector<Atom*> silanes,hydrogens;//hToRem;
    vector<atomT*> h;
    h.push_back(aTypes[7]);
    getAtomsByType(hydrogens,h);
    int functHere;
    while (count>0) {
        cout << count << " molecules to append" << endl;
        functHere = remoteH(hydrogens,silanes);
        //cout << "H supposedly found, " << functHere << ", Appending" << endl;
        silanes.push_back( appendMol(hydrogens[functHere],moly));
        //hToRem.push_back(*functHere);
        hydrogens.erase(hydrogens.begin()+functHere);
        count--;
    }
    silaneBridge();
    dim[5]=top()+2.0;
    perDim[2]=dim[5]-dim[4];
    checkPos();
    verifyCon(0);
    checkOs();
    compressAll();
    atoms.fitDimension(dim,perDim);
}

void Particle::readSilane(istream& in) {
    stringstream ms,ps,bs,as,ds;
    ms << 8 << ' ' << 27.9769 << endl;
    ms << 9 << ' ' << 12 << endl;
    ms << 10 << ' ' << 15.9949 << endl;
    ms << 11 << ' ' << 12 << endl;
    ms << 12 << ' ' << 1.00782 << endl;
    ps << "8 0.1 4.0" << endl;
    ps << "9 0.066 3.5" << endl;
    ps << "10 0.17 3.0" << endl;
    ps << "11 0.066 3.5" << endl;
    ps << "12 0.03 2.5" << endl;
    bs << "3 200 1.850" << endl;
    bs << "4 268 1.529" << endl;
    bs << "5 340 1.090" << endl;
    as << "4 23.7764 122.888" << endl;
    as << "5 37.5 110.7" << endl;
    as << "6 60 100 " << endl;
    as << "7 58.35 112.7" << endl;
    as << "8 37.5 110.7" << endl;
    as << "9 33 107.8" << endl;
    ds << "1 1.74 -0.157 0.279 0" << endl;
    ds << "2 0 0 0.366 0" << endl;
    ds << "3 0 0 0.318 0" << endl;
    aTypes.readMass(ms,5);
    aTypes.readPairs(ps);
    bTypes.readBTypes(bs,3);
    angTypes.readAngTypes(as,6);
    dTypes.readDTypes(ds,3);
    string line;
    while (!in.eof()) {
        getline(in,line);
        if (line.find("ATOM")!=string::npos) {
```

```
                break;
        }
    }
    stringstream ATOMS;
    line.clear();
    //inputs for atoms
    string type;
    FLOAT x, y, z;
    int index;
    while (!in.eof()) {
        if (in.peek()=='@')
                break;
        in >> index >> type >> x >> y >> z;
        ATOMS << index << ' ';
        if (type.find("C")!=string::npos) {
                ATOMS << 1 << ' ' <<  9 << ' ' << -0.120 << ' ';
        } else if (type.find("H")!=string::npos) {
                ATOMS << 1 << ' ' << 12 << ' ' << 0.060 << ' ';
        } else if (type.find("SI")!=string::npos) {
                ATOMS << 1 << ' ' <<  8 << ' ' << 0.745 << ' ';
        } else
                cerr << "Couldn't type mol2 atom";
        ATOMS << x << ' ' << y << ' ' << z << " 0 0 0" << endl;
        getline(in,line);
    }
    atoms.readAtoms(ATOMS,index);
    getline(in,line); //move to line past bond identifier
    stringstream BONDS; //stringstream to store bonds
    atomT* Si=aTypes[8];
    atomT* C=aTypes[9];
    atomT* H=aTypes[12];
    unsigned int atm1, atm2;
    int t;
    while (!in.eof()) {
        if (in.peek()=='@')
                break;
        in >> index >> atm1 >> atm2;
        getline(in,line);
        if (in.eof())
                break;
        Atom* A1=atoms.atmInd[atm1];
        Atom* A2=atoms.atmInd[atm2];
        if (A1->type==H || A2->type==H)
                t=5;
        else if (A1->type==C && A2->type==C)
                t=4;
        else
                t=3;
        BONDS << index << ' ' << t << ' ' << atm1 << ' ' << atm2 << endl;
    }
    bonds.readBonds(BONDS,index);
    int b,h=0; vector<Atom*> bonded;
    //retype the terminal methyl
    for (int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL) continue;
        if (atoms[i]->type!=C) continue;
        b=atoms[i]->getBonded(bonded);
        for (int j=0; j<b; ++j) {
                if (bonded[j]->type==H)
                        h++;
        }
        if (h==3) {
                atoms[i]->type=aTypes[11];
                atoms[i]->charge=-0.180;
                break;
```

```
        } else {
                bonded.clear(); h=0;
        }
    }
    vector<Angle*> angs;
    angles.parseForSil(angs);
    dihedrals.parseForSil(angs);
}

int Particle::remoteHRound(vector<Atom*> &hydrogens, vector<Atom*> &silanes) {
    int s = silanes.size();
    int h = hydrogens.size();
    int i;
    if (silanes.size() == 0)
        return 0;
    int stride2 = sizeof(FLOAT);
    int stride0 = 3 * stride2;
    FLOAT **hPos = (FLOAT**)malloc(sizeof(FLOAT*) * h); //f
    FLOAT **sPos = (FLOAT**)malloc(sizeof(FLOAT*) * s); //f
    for (i = 0; i < h; ++i)
        hPos[i] = hydrogens[i]->getPos();
    for (i = 0; i < s; ++i)
        sPos[i] = silanes[i]->getPos();
    //I want the h which has the maximum possible angel from the silanes
    //First need normalized vectors
    //calculated norms
    FLOAT *hPosN = IPPF1(ippsMalloc)(h); //f
    FLOAT *sPosN = IPPF1(ippsMalloc)(s); //f
    IPPF2(ippmL2Norm_va,L)(hPos,0,stride2,hPosN,3,h);
    IPPF2(ippmL2Norm_va,L)(sPos,0,stride2,sPosN,3,s);
    //normalize vectors
    FLOAT *hPosVN = IPPF1(ippsMalloc)(3*h);
    FLOAT *sPosVN = IPPF1(ippsMalloc)(3*s);
    for (i = 0; i < h; ++i) {
        hPosVN[3*i + 0] = hPos[i][0] / hPosN[i];
        hPosVN[3*i + 1] = hPos[i][1] / hPosN[i];
        hPosVN[3*i + 2] = hPos[i][2] / hPosN[i];
    }
    free(hPos);
    for (i = 0; i < s; ++i) {
        sPosVN[3*i + 0] = sPos[i][0] / sPosN[i];
        sPosVN[3*i + 1] = sPos[i][1] / sPosN[i];
        sPosVN[3*i + 2] = sPos[i][2] / sPosN[i];
    }
    ippsFree(hPosN); ippsFree(sPosN);
    free(sPos);
    FLOAT *dots = IPPF1(ippsMalloc)(h*s);
    for (i = 0; i < h; ++i ) {
        IPPF1(ippmDotProduct_vav)(sPosVN,stride0,stride2,hPosVN+3*i,stride2,dots+i*s,3,s);
    }
    FLOAT max, *dotPos;
    FLOAT *maxs = IPPF1(ippsMalloc)(h);
    int j;
    for (i = 0; i < h; ++i) {
        max = -1;
        dotPos = dots + s*i;
        for (j = 0; j < s; ++j) {
                if (dotPos[j]>max)
                        max = dotPos[j];
        }
        maxs[i] = max;
    }
    ippsFree(dots);
    FLOAT min = 0;
    int minInd;
```

```
    IPPF1(ippsMinIndx)(maxs,h,&min,&minInd);
    ippsFree(maxs);
    return minInd;
}

int Particle::remoteHFlat(vector<Atom*> &hydrogens, vector<Atom*> &silanes) {
    int s=silanes.size(), h=hydrogens.size();
    if (silanes.size()==0)
        return 0;
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
    #else
        //TODO non-ipp branch
    #endif
    FLOAT** hPos=(FLOAT**)malloc(sizeof(FLOAT*)*hydrogens.size());
    FLOAT** sPos=(FLOAT**)malloc(sizeof(FLOAT*)*silanes.size());
    for (int i=0; i<hydrogens.size(); ++i)
        hPos[i]=hydrogens[i]->getPos();
    for (int i=0; i<silanes.size(); ++i)
        sPos[i]=silanes[i]->getPos();
    FLOAT rel[3],dist,min[hydrogens.size()];
    for (int i=0; i<hydrogens.size(); ++i) {
        min[i]=1000000000.0;
        for (int j=0; j<silanes.size(); ++j) {
                rel[0]=hPos[i][0]-sPos[j][0];
                rel[1]=hPos[i][1]-sPos[j][1];
                rel[2]=hPos[i][2]-sPos[j][2];
                for (int k=0; k<3; ++k) {
                        if (fabs(rel[k]) > perDim[k]/2.0) {
                                if (rel[k]>0)
                                        rel[k]-=perDim[k];
                                else
                                        rel[k]+=perDim[k];
                        }
                }
                if (periodic[0]==1 && periodic[1]==1 && periodic[2]==0)
                        dist=sqrt(rel[0]*rel[0]+rel[1]*rel[1]);
                else
                        dist=sqrt(rel[0]*rel[0]+rel[1]*rel[1]+rel[2]*rel[2]);
                if (dist<min[i])
                        min[i]=dist;
        }
    }
    FLOAT max=0;
    int maxInd;
    #ifdef ipp
        IPPF1(ippsMaxIndx)(min,hydrogens.size(),&max,&maxInd);
    #endif
    free(hPos);
    free(sPos);
    return maxInd;
}

int Particle::remoteH(vector<Atom*> &hydrogens, vector<Atom*> &silanes) {
    if (periodic[0] && periodic[1])
        return remoteHFlat(hydrogens,silanes);
    else
        return remoteHRound(hydrogens,silanes);
}

Atom* Particle::appendMol(Atom* appHere, Particle &moly) {
    //cout << "Appending Function" << endl;
    vector<Atom*> hbonds;
    int b=appHere->getBonded(hbonds);
```

```
    if (b>1) {
        DEB(H has too many atoms)
        abort;
    }
    //Get the O that will be bound to the new molecule, use its position for relocation of the
    molecule
    Atom* O=hbonds[0];
    O->type=aTypes[10];
    FLOAT* oPos=O->getPos();
    atoms.remElem(appHere); //get rid of the hydrogen
    //make new molecule and transfer
    Molecule *newM=new Molecule;
    Molecule *oldM=moly.molecules[0];
    transMol(*oldM, *newM, 0); //last argument instructs transmol not to delete the preexisting
    molecule
    molecules.addElem(newM); //add to list
    //relocate the new molecule appropriately
    vector<Atom*> oBonded;
    b=O->getBonded(oBonded);
    if (b>1) {
        DEB(H has too many atoms)
        abort();
    }
    FLOAT *basePos=oBonded[0]->getPos();
    if (periodic[0]==1 && periodic[1]==1 && periodic[2]==0) {
        FLOAT newLoc[3];
        oPos[0]=basePos[0];
        oPos[1]=basePos[1];
        oPos[2]=basePos[2]+1.6;
        newLoc[0]=oPos[0];
        newLoc[1]=oPos[1];
        newLoc[2]=oPos[2]+1.6;
        newM->translate(newLoc); //translate to new location
    } else {
        FLOAT rot[3],R;
        rot[0] = basePos[0];
        rot[1] = basePos[1];
        rot[2] = basePos[2];
        R = sqrt(rot[0]*rot[0]+rot[1]*rot[1]+rot[2]*rot[2]);
        rot[0]/=R; rot[1]/=R; rot[2]/=R;
        newM->rotate(rot);
        FLOAT newpos[3];
        newpos[0] = basePos[0]+rot[0]*1.6;
        newpos[1] = basePos[1]+rot[1]*1.6;
        newpos[2] = basePos[2]+rot[2]*1.6;
        newM->translate(newpos);
    }
    //bind the molecule
    Atom* silane, *firstC;
    newM->findSilaneEnd(silane,firstC);
    bondT *OSi=bTypes[1];
    angleT *OSiC=angTypes[6];
    angleT *SiOSi=angTypes[1];
    bonds.addBond(O,silane,OSi);
    angles.addAngle(O,silane,firstC,OSiC);
    angles.addAngle(silane,O,oBonded[0],SiOSi);
    //cout << "Appended Function" << endl;
    return silane;
}

void Particle::silaneBridge() {
    Molecule* bulk;
    for (int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL) continue;
        if (aTypes[atoms[i]->type]<8) {
```

```
            bulk=atoms[i]->mol;
            break;
    }
}
list<Atom*> silanes, silicons, connected;
vector<atomT*> silTypes;
silTypes.push_back(aTypes[8]);
getAtomsByType(silanes,silTypes);
silTypes.push_back(aTypes[3]);
silTypes.push_back(aTypes[5]);
getAtomsByType(silicons,silTypes);
atomT* surfO=aTypes[6];
atomT* silaType=aTypes[8];
while (silanes.size()>0) {
    cout << silanes.size() << " silanes left to finalize" << endl;
    vector<Atom*> bridgeSet, siBonds, forbidden; //bridgeSet will store atoms that can be
bridged
    int b=(*silanes.begin())->getBonded(siBonds); //gets bonds of silane
    bridgeSet.push_back(*silanes.begin()); //pushes silane to top of list
    for (int i=0; i<b; ++i) {
            if (siBonds[i]->type->mass==15.9949) {
                    vector<Atom*> bonded2;
                    int b2=siBonds[i]->getBonded(bonded2);
                    if (bonded2[0]!=(*silanes.begin())) {
                            forbidden.push_back(bonded2[0]);
                    } else if (bonded2[1]!=(*silanes.begin())) {
                            forbidden.push_back(bonded2[1]);
                    } //forbid bridging that will form Si*-O-Si-O-Si* loops
            }
    }
    for (list<Atom*>::iterator i=silicons.begin(); i!=silicons.end(); ++i) { //looping
through silicons to find appropriate bridgers
            if ((*i)==(*silanes.begin())) continue; //ignore the silane of interest
            if ((*i)->type->mass!=27.9769) continue; //ignore anything that might have come
through with the wrong mass
            vector<Atom*> bonded2;
            int b2=(*i)->getBonded(bonded2);  //Get things bonded to silicon of interest
            if (b2==4) {  //if it has 4 bonds, then check it out, otherwise I guess maybe
it's an internal and can be ignored
                    bool check=0;
                    for (int j=0; j<b2; ++j) {
                            if (bonded2[j]->type==surfO) { //if this silicon is bound to a
surface oxygen, then we're interested in it
                                    check=1;
                                    bridgeSet.push_back(*i);
                                    break;
                            }
                    }
                    if (check) continue;
            }
            if (b2==4 && (*i)->type==silaType) { //Also if this silicon is a silane then
we're interested in it
                    bridgeSet.push_back(*i);
            }
    }
    int* indices=(int*)malloc(sizeof(int)*(bridgeSet.size()-1)); //stores the call back
indices for the sorted distances
    #ifdef ipp
            FLOAT* N=IPPF1(ippsMalloc)(bridgeSet.size()-1);
    #else
            FLOAT* N=(FLOAT*)malloc(sizeof(FLOAT)*bridgeSet.size()-1);
    #endif
    closestToFirst(bridgeSet,indices,N,periodic,perDim); //the silane is first entry, the
rest are possible binders, gives sorted distances N and call back indices for bridgeSet (add
1)
```

```
    int I=0;
    while ((*silanes.begin())->bonds.size()<4 && N[I]<6.5) { //iterate so long as distance is
less than 5.5, after that the thing hydroxylates
            bool check=0;
            Atom* SiOI=bridgeSet[indices[I]+1]; //silicon of interest
            Atom* sil=bridgeSet[0]; //silane
            for (int i=0; i<forbidden.size(); i++) {
                    if (SiOI==forbidden[i]) { //if this entry is forbidden, move along
                            ++I;
                            check=1;
                            break;
                    }
            }
            if (check) continue;
            if (SiOI->type==aTypes[8]) { //its a silane, so do a 2-atom bridge(i.e. need to
make the oxygen), it wouldn't be hydroxylated because it would've found this one...
                    if (SiOI->bonds.size()<4)
                            makeBridge(sil,SiOI,bulk);
                    ++I;
                    continue;
            } else {
                    vector<Atom*> bonds;
                    SiOI->getBonded(bonds); //get atoms bonded to this silicon
                    check=0;
                    for (int j=0; j<bonds.size(); ++j) {
                            if (bonds[j]->type==aTypes[6]) { //if it binds a hydroxyl
oxygen, time to have fun
                                    vector<Atom*> oBonds;
                                    int ob=bonds[j]->getBonded(oBonds);      //get bonds to
the oxygen
                                    if (oBonds[0]->type==aTypes[7]) { //if the oxygen is
bound to a 7, it's a hydroxyl, time for funzies
                                            forbidden.push_back(SiOI);
                                            atoms.remElem(oBonds[0]->getInd()); //take out
the attached hydrogen
                                            makeBridge(SiOI,bonds[j],sil); //form three
bond, bridgeSet is the silicon of interest, bonds is the oxygen, finally the silane of
interest
                                            check=1;
                                            ++I;
                                            break;
                                    } else if (oBonds[1]->type==aTypes[7]) { //same as
previous statement, just checking the other bond of the O
                                            forbidden.push_back(SiOI);
                                            atoms.remElem(oBonds[1]->getInd());
                                            makeBridge(SiOI,bonds[j],sil);
                                            check=1;
                                            ++I;
                                            break;
                                    } else {
                                            cerr << "out of place hydroxyl O type" << endl;
//no hydrogen was found, this is not a hydroxyl oxygen, bah! fix it and complain
                                            bonds[j]->type=aTypes[4];
                                            bonds[j]->charge=-0.430;
                                    }
                            }
                    }
                    if (!check) {
                            ++I;
                    }
            }
    }
    addHydroxyls(*silanes.begin(),bulk);
    silanes.erase(silanes.begin());
}
```

```
}

void Particle::makeBridge(Atom* si1, Atom *si2, Molecule *bulk) {

    atomT* sil=aTypes[8]; //silane type
    atomT* brO=aTypes[10]; //bridging oxygen type
    atomT* sili=aTypes[5]; //surface silicon type
    FLOAT* si1P=si1->getPos();
    FLOAT* si2P=si2->getPos();
    Atom* newO=atoms.createNew();
    newO->charge=-0.215;
    newO->type=brO;
    bulk->addAtom(newO);
    FLOAT* oPos=newO->getPos();
    setMidp(oPos, si1P, si2P, periodic, perDim);
    vector<Atom*> si1Bnds, si2Bnds;
    int si1BC=si1->getBonded(si1Bnds);
    int si2BC=si2->getBonded(si2Bnds);
    Bond* b1=bonds.addBond(si1,newO,bTypes[1]);
    Bond* b2=bonds.addBond(si2,newO,bTypes[1]);
    Angle* SiOSi=angles.addAngle(si1,newO,si2,angTypes[1]);
    for (int i=0; i<si1BC; ++i ) {
        Angle *newAng=angles.addAngle(si1Bnds[i],si1,newO,NULL);
        if (si1Bnds[i]->type->mass==15.9949)
                newAng->type=angTypes[2];
        else if (si1Bnds[i]->type->mass==12.0)
                newAng->type=angTypes[6];
    }
    for (int i=0; i<si2BC; ++i ) {
        Angle *newAng=angles.addAngle(si2Bnds[i],si2,newO,NULL);
        if (si2Bnds[i]->type->mass==15.9949)
                newAng->type=angTypes[2];
        else if (si2Bnds[i]->type->mass==12.0)
                newAng->type=angTypes[6];
    }
}

void Particle::makeBridge(Atom* si1, Atom* o, Atom* si2) { //assumes si2 is the silane, so that
    the O-Si1-X angles are presumed taken care of
    atomT* sil=aTypes[8];
    atomT* brO=aTypes[10];
    atomT* sili=aTypes[5];
    FLOAT* si1P=si1->getPos();
    FLOAT* si2P=si2->getPos();
    FLOAT* oPos=o->getPos();
    setMidp(oPos, si1P, si2P, periodic, perDim);
    vector<Atom*> si1Bnds, si2Bnds;
    int si1BC=si1->getBonded(si1Bnds);
    int si2BC=si2->getBonded(si2Bnds);
    //Bond* b1=bonds.addBond(si1,o,bTypes[1]);
    Bond* b2=bonds.addBond(si2,o,bTypes[1]);
    Angle* SiOSi=angles.addAngle(si1,o,si2,angTypes[1]);
    for (int i=0; i<si2BC; ++i ) {
        Angle *newAng=angles.addAngle(si2Bnds[i],si2,o,NULL);
        if (si2Bnds[i]->type->mass==15.9949)
                newAng->type=angTypes[2];
        else if (si2Bnds[i]->type->mass==12.0)
                newAng->type=angTypes[6];
    }
}

void Particle::addHydroxyls(Atom* sil, Molecule *bulk) {
    vector<Atom*> siBonds;
    int bc=sil->getBonded(siBonds);
    if (bc==4)
```

```
        return;
    int c;
    atomT* brO=aTypes[10];
    atomT* HO=aTypes[7];
    atomT* surfO=aTypes[4];
    atomT* Oh=aTypes[6];
    atomT* C=aTypes[9];
    bondT* SiO=bTypes[1];
    bondT* OH=bTypes[2];
    angleT* SiOH=angTypes[3];
    angleT* CSiO=angTypes[6];
    angleT* OSiO=angTypes[2];
    FLOAT* silPos=sil->getPos();
    c=4-bc;
    while (c>0) {
        Atom* newO=atoms.createNew();
        Atom* newH=atoms.createNew();
        newO->type=Oh;
        newO->charge=-0.683;
        newH->type=HO;
        newH->charge=0.418;
        bulk->addAtom(newO);
        bulk->addAtom(newH);
        Bond* SiOBnd=bonds.addBond(sil,newO,SiO);
        Bond* OHBnd=bonds.addBond(newO,newH,OH);
        Angle* SiOHAng=angles.addAngle(sil,newO,newH,SiOH);
        for (int i=0; i<siBonds.size(); ++i) {
                if (siBonds[i]->type==brO || siBonds[i]->type==surfO || siBonds[i]->type==Oh)
                        angles.addAngle(siBonds[i],sil,newO,OSiO);
                else if (siBonds[i]->type==C)
                        angles.addAngle(siBonds[i],sil,newO,CSiO);
                else
                        cerr << "Can't type angle for X-Si-O during hydroxylation" << endl;
        }
        FLOAT* nhPos=newH->getPos();
        FLOAT* noPos=newO->getPos();
        if (periodic[0] && periodic[1]) {
                nhPos[2]=silPos[2];
                noPos[2]=silPos[2];
                if (c==2) {
                        noPos[0]=silPos[0]+1.6;
                        noPos[1]=silPos[1];
                        nhPos[0]=silPos[0]+2.6;
                        nhPos[1]=silPos[1];
                } else if (c==1) {
                        noPos[0]=silPos[0];
                        noPos[1]=silPos[1]+1.6;
                        nhPos[0]=silPos[0];
                        nhPos[1]=silPos[1]+2.6;
                }
        } else {
                FLOAT silPosDir[3], silPosR;
                silPosR = sqrt(silPos[0]*silPos[0]+silPos[1]*silPos[1]+silPos[2]*silPos[2]);
                silPosDir[0] = silPos[0]/silPosR;
                silPosDir[1] = silPos[1]/silPosR;
                silPosDir[2] = silPos[2]/silPosR;
                FLOAT newDir[3], ndr;
                newDir[0] = silPosDir[1]/silPosDir[0];
                newDir[2] = 0;
                if (c==2) {
                        newDir[1] = -1;
                } else if (c==1) {
                        newDir[1] = 1;
                }
                ndr = sqrt(1+newDir[0]*newDir[0]);
```

```
                newDir[0]/=ndr;
                newDir[1]/=ndr;
                noPos[0] = silPos[0] + (1.6 * newDir[0]);
                noPos[1] = silPos[1] + (1.6 * newDir[1]);
                noPos[2] = silPos[2];
                nhPos[0] = silPos[0] + (2.6 * newDir[0]);
                nhPos[1] = silPos[1] + (2.6 * newDir[0]);
                nhPos[2] = silPos[2];
        }
        c--;
        siBonds.push_back(newO);
    }
}

FLOAT Particle::top() {
    FLOAT max=0;
    FLOAT *pos;
    for (int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL) continue;
        pos=atoms[i]->getPos();
        if (pos[2]>max)
                max=pos[2];
    }
    return max;
}

void Particle::checkOs() {
    vector<atomT*> ots;
    atomT* H=aTypes[7];
    atomT* surfO=aTypes[4];
    atomT* OH=aTypes[6];
    ots.push_back(OH);
    vector<Atom*> os;
    getAtomsByType(os,ots);
    int b,j;
    bool check;
    for (int i=0; i<os.size(); ++i) {
        vector<Atom*> bonds;
        b=os[i]->getBonded(bonds);
        check=0;
        for (j=0;j<b;++j) {
                if (bonds[j]->type==H) {
                        check=1;
                        break;
                }
        }
        if (!check) {
                os[i]->type=surfO;
                os[i]->charge=-0.430;
                cerr << "Hydroxyl O flipped to surface O" << endl;
        }
    }
    ots.clear();
    ots.push_back(surfO);
    os.clear();
    getAtomsByType(os,ots);
    for (int i=0; i<os.size(); ++i) {
        vector<Atom*> bonds;
        b=os[i]->getBonded(bonds);
        check=0;
        for (j=0; j<b; ++j) {
                if (bonds[j]->type==H) {
                        check=1;
                        break;
                }
```

```
        }
        if (check) {
                os[i]->type=OH;
                os[i]->charge=-0.683;
                cerr << "Surface O flipped to OH" << endl;
        }
    }
    ots.clear();
    ots.push_back(aTypes[8]);
    os.clear();
    getAtomsByType(os,ots);
    for (int i=0; i<os.size(); ++i) {
        if (os[i]->bonds.size()!=4) {
                DEB(Silane without 4 bonds)
        }
    }
}

void Particle::checkPos() {
    FLOAT *aPos;
    for (int i=0; i<3; ++i) {
        if (periodic[i]) {
                perDim[i]=dim[2*i+1]-dim[2*i];
                for (int j=0; j<atoms.max(); ++j) {
                        if (atoms[j]==NULL) continue;
                        aPos=atoms[j]->getPos();
                        while (aPos[i]>dim[2*i+1])
                                aPos[i]-=perDim[i];
                        while (aPos[i]<dim[2*i])
                                aPos[i]+=perDim[i];
                }
        }
    }
}
```

Particle Coverage Calculation Functions **ParticleCovCalc.cpp**

```
#ifndef ipp
    #include "bMathLibs.h"
#endif
#include <omp.h>
#include "inc.h"

void Particle::calcCoverageProps(ostream& covOut, ostream& covMol2, ostream& distOut, ostream&
    dirOut,  FLOAT radius, FLOAT angle, FLOAT meshRad) {
    if (periodic[0]==1 && periodic[1]==1 && periodic[2]==0) {
        cout << "Calculating film properties for flat system" << endl;
        list<Atom*> hydrocarbs = calcCovAllFlat(covOut,angle,meshRad);
        calcRaDistFlat(distOut,hydrocarbs);
        calcDirections(dirOut);
    } else {
        cout << "Calculating film properties for round system" << endl;
        FLOAT* centers;
        list<Atom*> hydrocarbs = calcCovAllRound(covOut,covMol2,radius,angle,centers,meshRad);
        calcRaDistRound(distOut,centers,hydrocarbs);
        calcDirections(dirOut,centers);
    }
}

list<Atom*> Particle::calcCovAllFlat(ostream& out, FLOAT angle, FLOAT meshRad) {
    cout << "Calculating Coverage for flat system" << endl;
    FLOAT cut=cos(angle/R2D);
    vector<atomT*> typeComp;
    typeComp.push_back(aTypes[9]);
```

```
    typeComp.push_back(aTypes[11]);
    typeComp.push_back(aTypes[12]);
    list<Atom*> hydrocarbs;
    getAtomsByType(hydrocarbs,typeComp);
    #ifdef ipp
        FLOAT* heights=IPPF1(ippsMalloc)(dumps);
    #else
        FLOAT* heights=(FLOAT*)malloc(sizeof(FLOAT)*dumps);
    #endif
    getFlatDumpHeights(heights);
    FLOAT* points;
    int N=rectGrid(dim,meshRad,heights,dumps,points);
    bool isCov[dumps*N];
    #ifdef ipp
        FLOAT* dir=IPPF1(ippsMalloc)(3);
        dir[0]=0; dir[1]=0; dir[2]=1;
    #else
        FLOAT dir[3]={0,0,1};
    #endif
    #pragma omp parallel for schedule(dynamic)
    for (int i=0; i<N; ++i) {
        vector<Atom*> neighbs;
        parseByDist(hydrocarbs,neighbs,points+3*dumps*i,40.0);
        checkCov(points+3*dumps*i,dir,neighbs,isCov+dumps*i,cut,1,dumps);
        int n=omp_get_thread_num();
        if (n==0) {
                cout << "Point " << i << " calculated, " << N << " total" << endl;
        }
    }
    FLOAT covs[dumps];
    #pragma omp parallel for
    for (int i=0; i<dumps; ++i) {
        int pointCount=0;
        for (int j=0; j<N; ++j) {
                if (isCov[j*dumps+i])
                        ++pointCount;
        }
        cout << pointCount << ' ' << N << endl;
        covs[i]=1.0-FLOAT(pointCount)/FLOAT(N);
    }
    for (int i=0; i<dumps; i++) {
        cout << covs[i] << endl;
        out << setw(10) << left << timesteps[i] << setw(10) << setprecision(8) << left <<
    100.0*covs[i] << endl;
    }
    #ifdef ipp
        ippsFree(points);
        ippsFree(heights);
        ippsFree(dir);
    #else
        free(heights);
    #endif
    return hydrocarbs;
}

void Particle::getFlatDumpHeights(FLOAT* out) {
    vector<atomT*> silT;
    silT.push_back(aTypes[8]);
    list<Atom*> silanes;
    getAtomsByType(silanes,silT);
    FLOAT* silPos[silanes.size()];
    int j=0;
    for (list<Atom*>::iterator i=silanes.begin(); i!=silanes.end(); ++i,++j) {
        silPos[j]=(*i)->getPos(1);
    }
```

```
    #pragma omp parallel for
    for (int i=0; i<dumps; ++i) {
        #ifdef ipp
                FLOAT* dpos=IPPF1(ippsMalloc)(silanes.size());
                for (int j=0; j<silanes.size(); ++j)
                        dpos[j]=silPos[j][3*i+2];
                IPPF1(ippsMean)(dpos,silanes.size(),out+i);
                ippsFree(dpos);
        #else
                out[i]=0;
                for (int j=0; j<silanes.size(); ++j)
                        out[i]+=silPos[j][3*i+2];
                out[i]/=silanes.size();
        #endif
    }
}

list<Atom*> Particle::calcCovAllRound(ostream& out, ostream& covMol2, FLOAT radius, FLOAT angle,
    FLOAT*& centers, FLOAT meshRad) {
    //Figure out how many points are needed based on the radius and mesh radius, mesh radius the
    desired half distance between points on the sphere
    FLOAT cut=cos(angle/R2D);
    FLOAT area=4*PI*pow(radius,2);
    FLOAT meshArea=PI*pow(meshRad,2);
    FLOAT p=area/meshArea;
    int N=int(ceil(p));
    //First collect silane atoms as these dictate where the centers will be positions
    vector<Atom*> silAtoms;
    getSilAtoms(silAtoms);
    #ifdef ipp
        centers=IPPF1(ippsMalloc)(3*dumps);  //Storage is [center dump1, center dump2, ... center
    dumpn]
        FLOAT* points=IPPF1(ippsMalloc)(3*dumps*N); //Storage is [ [p1dump1,p1dump2,...]
    [p2dump1,p2dump2,...]...[pndump1,...pndumpn]]
        FLOAT* nPoints=IPPF1(ippsMalloc)(3*N);      //Storage is [ p1,p2,...pn ]  These are
    directional and therefore need not take the center into account
    #else
        centers=(FLOAT*)malloc(3*dumps*sizeof(FLOAT));
        FLOAT points[3*dumps*N];
        FLOAT nPoints[3*dumps*N];
    #endif
    //Calculates the centers based on silanes, calculates centers for every dump
    FLOAT R=calcCenter(silAtoms,centers,1,dumps);
    radSpir(centers,R,dumps,N,points,nPoints);
    //Next collect the relevant atoms at each point, this uses the first dump and gathers all
    hydrocarbon atoms within 4 nm
    list<Atom*> hydrocarbs;
    vector<atomT*> typeComp;
    typeComp.push_back(aTypes[9]);
    typeComp.push_back(aTypes[11]);
    typeComp.push_back(aTypes[12]);
    getAtomsByType(hydrocarbs,typeComp);
    DEB(Atoms Collected)
    //Calculated coverages
    bool isCov[dumps*N];  //Storage is [
    [p1d1,p1d2...p1dn],[p2d1,p2d2,...p2dn],...[pNd1,pNd2,...pNdn]]
    #pragma omp parallel for schedule(dynamic)
    for (int i=0; i<N; ++i) {
        vector<Atom*> neighbs;
        parseByDist(hydrocarbs,neighbs,points+3*dumps*i,40.0);
        checkCov(points+3*dumps*i,nPoints+3*i,neighbs,isCov+dumps*i,cut,1,dumps);
        int n=omp_get_thread_num();
        if (n==0) {
                cout << "Point " << i << " calculated, " << N << " total" << endl;
        }
```

```
    }
    DEB(Cov Determined)
    FLOAT covs[dumps];
    #pragma omp parallel for
    for (int i=0; i<dumps; ++i) {
        int pointCount=0;
        for (int j=0; j<N; ++j) {
                if (isCov[j*dumps+i])
                        ++pointCount;
        }
        //cout << pointCount << ' ' << N << endl;
        covs[i]=1.0-FLOAT(pointCount)/FLOAT(N);
    }
    printCovMol2(covMol2,points,isCov,dumps,N);
    for (int i=0; i<dumps; i++) {
        //cout << "Test " << covs[i] << endl;
        out << setw(10) << left << timesteps[i] << setprecision(8) << 100.0*covs[i] << endl;
    }
    #ifdef ipp
        ippsFree(points);
        ippsFree(nPoints);
    #endif
    return hydrocarbs;
}

void Particle::checkCov(FLOAT* point, FLOAT* pointDir, vector<Atom*>& relAtoms, bool* flags,
    FLOAT cut, int s, int d) {
    int j=relAtoms.size();
    #ifdef ipp
        FLOAT* rels=IPPF1(ippsMalloc)(3*d*j);
    #else
        FLOAT* rels=(FLOAT*)malloc(3*d*j*sizeof(FLOAT));
    #endif
    FLOAT** pos=(FLOAT**)malloc(j*sizeof(FLOAT));
    FLOAT** pPoint=(FLOAT**)malloc(j*sizeof(FLOAT));
    FLOAT** pRel=(FLOAT**)malloc(j*sizeof(FLOAT));
    for (int i=0; i<j; ++i) {
        pos[i]=relAtoms[i]->getPos(s);
        pRel[i]=rels+3*d*i;
    }
    #ifdef ipp
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        IPPF2(ippmSub_vav,L)(pos,0,stride2,point,stride2,pRel,0,stride2,3*d,j);
        periodicCheck(rels,1,dumps,j);
        normalizeVecs(rels,j*d);
        FLOAT* dots=IPPF1(ippsMalloc)(j*d);
        IPPF1(ippmDotProduct_vav)(rels,stride0,stride2,pointDir,stride2,dots,3,j*d);
        ippsFree(rels);
    #else
        sub_vavL(pos,point,pRel,3*d,j);
        periodicCheck(rels,1,dumps,j);
        normalizeVecs(rels,j*d);
        FLOAT* dots=(FLOAT*)malloc(j*d*sizeof(FLOAT));
        dotP_vav(rels,pointDir,dots,3,j*d);
        free(rels);
    #endif
    for (int i=0; i<d; ++i) {
        flags[i]=0;
        for (int k=0; k<j; k++) {
                if (dots[d*k+i]>cut) {
                        flags[i]=1;
                        break;
                }
        }
```

```
    }
    free(pos);
    free(pRel);
    free(pPoint);
    #ifdef ipp
        ippsFree(dots);
    #else
        free(dots);
    #endif
}

void Particle::calcRaDistFlat(ostream& out, list<Atom*>& hydrocarbs) {
    cout << "Calculating Hydrocarbon Distribution for Round Particle" << endl;
    unsigned int hcs=hydrocarbs.size();
    #ifdef ipp
        FLOAT* pos=IPPF1(ippsMalloc)(hcs*dumps);
    #else
        FLOAT* pos=(FLOAT*)malloc(sizeof(FLOAT)*hcs*dumps);
    #endif
    int k=0;
    for (list<Atom*>::iterator i=hydrocarbs.begin(); i!=hydrocarbs.end(); ++i, ++k) {
        FLOAT* tpos=(*i)->getPos(1);
        for (int j=0; j<dumps; ++j)
                pos[hcs*j+k]=tpos[3*j+2];
    }
    cout << "Collecting and sorting positions" << endl;
    #ifdef ipp
        FLOAT min, max, slope;
        IPPF1(ippsMinMax)(pos,hydrocarbs.size()*dumps,&min,&max);
        min-=1.0;
        max+=1.0;
        slope=(max-min)/100.0;
        FLOAT* bin=IPPF1(ippsMalloc)(100);
        FLOAT* histograms=IPPF1(ippsMalloc)(100*dumps);
        IPPF1(ippsVectorSlope)(bin,100,min,slope);
        #pragma omp parallel for
        for (int i=0; i<dumps; ++i)
                IPPF2(ippsSortAscend,I)(pos+hcs*i,hcs);
    #else
        FLOAT min, max, slope;
        min=findMin(pos,hcs*dumps);
        max=findMax(pos,hcs*dumps);
        min-=1.0;
        max+=1.0;
        slope=(max-min)/100.0;
        FLOAT* bin=(FLOAT*)malloc(100*sizeof(FLOAT));
        FLOAT* histograms=(FLOAT*)malloc(100*dumps*sizeof(FLOAT));
        vectorSlope(min,slope,100,bin);
        #pragma omp parallel for
        for (int i=0; i<dumps; ++i)
                sort(pos+hcs*i,hcs);
    #endif
    FLOAT V=perDim[0]*perDim[1]*(bin[1]-bin[0]);
    cout << "Preparing histogram data" << endl;
    #pragma omp parallel for
    for (int i=0; i<dumps; ++i) {
        int total=0;
        FLOAT* p=pos+hcs*i;
        FLOAT* hist=histograms+100*i;
        for (int j=0; j<100; ++j) {
                hist[j]=0;
                if (total==hcs)
                        continue;
                while (p[total]<bin[j]) {
                        ++hist[j];
```

```
                        ++total;
                        if (total==hcs)
                                break;
                }
        }
    }
    cout << "Saving histogram data" << endl;
    for (int i=0; i<dumps; ++i) {
        out << "TIMESTEP: " << timesteps[i] << endl;
        for (int j=0; j<100; ++j) {
                out     << bin[j] << ' ' << FLOAT(histograms[100*i+j])/V << endl;
        }
    }
    #ifdef ipp
        ippsFree(bin);
        ippsFree(histograms);
        ippsFree(pos);
    #else
        free(bin);
        free(histograms);
        free(pos);
    #endif
}

void Particle::calcRaDistRound(ostream& out, FLOAT* centers, list<Atom*>& hydrocarbs) {
    cout << "Calculating Hydrocarbon Distribution for Round Particle" << endl;
    unsigned int n=hydrocarbs.size();
    #ifdef ipp
        FLOAT* hPos[n];
        //FLOAT* fromCent=IPPF1(ippsMalloc)(3*n*dumps);
        FLOAT **pFromCent=alloc2d(3*dumps,n);
        DEB(hist1)
        unsigned int j=0;
        int dumpStride=3*dumps;
        for (list<Atom*>::iterator i=hydrocarbs.begin(); i!=hydrocarbs.end(); ++i, ++j) {
                hPos[j]=(*i)->getPos(1);
        }
        DEB(hist2)
        int stride2=sizeof(FLOAT);
        int stride0=3*stride2;
        FLOAT* unsorted=IPPF1(ippsMalloc)(n*dumps);
        FLOAT* N=IPPF1(ippsMalloc)(n*dumps);
        DEB(hist3)
        //Note in the following the resort from atom sorted to dump sorted data
        //#pragma omp parallel for
        IPPF2(ippmSub_vav,L)(hPos,0,stride2,centers,stride2,pFromCent,0,stride2,dumpStride,n);
        DEB(hist3.5)
        for (int i=0; i<n; ++i) {
                IPPF1(ippmL2Norm_va)(pFromCent[i],stride0,stride2,unsorted+i*dumps,3,dumps);
        }
        DEB(hist4)
        free2d(pFromCent,n);
        IPPF1(ippmMul_vac)(unsorted,dumps*stride2,stride2,1.0,N,stride2,n*stride2,dumps,n);
        DEB(hist5)
        ippsFree(unsorted);
        FLOAT* histograms=IPPF1(ippsMalloc)(100*dumps);
        FLOAT* bin=IPPF1(ippsMalloc)(100);
        FLOAT* binSq=IPPF1(ippsMalloc)(100);
        FLOAT* iBinSq=IPPF1(ippsMalloc)(100);
        DEB(hist6)
        //Prepare bins
        FLOAT minRad, maxRad, slope;
        IPPF1(ippsMin)(N,n,&minRad);
        IPPF1(ippsMax)(N,n,&maxRad);
        minRad-=5; maxRad+=5;
```

```
        slope=(maxRad-minRad)/100;
        DEB(hist7)
        IPPF1(ippsVectorSlope)(bin,100,minRad,slope);
        IPPF1(ippsSqr)(bin,binSq,100);
        FLOAT* Vs=IPPF1(ippsMalloc)(100);
        Vs[0]=binSq[0];
        IPPF1(ippsSub)(binSq,binSq+1,Vs+1,99);
        IPPF2(ippsMulC,I)(PI,Vs,100);
        DEB(hist8)
        IPPFIX(ippsInv)(Vs,iBinSq,100);
        ippsFree(binSq);
    #else
        FLOAT* hPos[n];
        FLOAT fromCent[dumps][n][3];
        FLOAT N[dumps*n];
        FLOAT minRad, maxRad, slope;
        FLOAT bin[100];
        FLOAT histograms[100*dumps];
        unsigned int j=0;
        for (list<Atom*>::iterator i=hydrocarbs.begin(); i!=hydrocarbs.end(); ++i, ++j)
                hPos[j]=(*i)->getPos(1);
        #pragma omp parallel for
        for (int i=0; i<n; ++i) {
                for (int j=0; j<dumps; j++) {
                        FLOAT fc[3];
                        for (int k=0; k<3; k++) {
                                fc[k]=hPos[i][3*j+k];
                        }
                        N[n*j+i]=sqrt(pow(fc[0],2)+pow(fc[1],2)+pow(fc[2],2));
                }
        }
        minRad=findMin(N,n);
        maxRad=findMax(N,n);
        minRad-=5; maxRad+=5;
        slope=(maxRad-minRad)/100;
        vectorSlope(minRad, slope, 100, bin);
    #endif
    cout << "Preparing histogram data" << endl;
    //calculate radial distrubtions
    #pragma omp parallel for
    for (int i=0; i<dumps; ++i) {
        FLOAT* norms=N+n*i;
        FLOAT* hist=histograms+100*i;
        #ifdef ipp
                IPPF2(ippsSortAscend,I)(norms,n);
        #else
                sort(norms,n);
        #endif
        int total=0;
        for (int j=0; j<100; ++j) {
                hist[j]=0;
                while (norms[total]<bin[j]) {
                        ++hist[j];
                        ++total;
                        if (total==n)
                                break;
                }
                if (total==n)
                        break;
        }
    }
    cout << "Adjusting data for radial nature" << endl;
    #ifdef ipp
        ippsFree(N);
        FLOAT* newH=IPPF1(ippsMalloc)(100*dumps);
```

```
        #pragma omp parallel for
        for (int i=0; i<dumps; ++i) {
                IPPF1(ippsMul)(histograms+100*i,iBinSq,newH+100*i,100);
        }
        ippsFree(histograms);
        ippsFree(iBinSq);
    #else
        FLOAT newH[100*dumps];
        for (int i=0; i<dumps; ++i) {
                for (int j=0; j<100; ++j) {
                        newH[100*i+j]=histograms[100*i+j]/pow(bin[j],2);
                }
        }
    #endif
    cout << "Outputting results" << endl;
    for (int i=0; i<dumps; ++i) {
        out << "TIMESTEP: " << timesteps[i] << endl;
        for (int j=0; j<100; ++j)
                out << setw(15) << setprecision(8) << left << bin[j] << setw(12) <<
    setprecision(8) << newH[100*i+j] << endl;
    }
}

FLOAT Particle::calcCenter(vector<Atom*>& ats, FLOAT* out, int s, int d) {
    int c=ats.size();
    if (c==0) {
        getSilAtoms(ats);
        c=ats.size();
    }
    //cout << "Finding center of " << c << " silane atoms" << endl;
    #ifdef ipp
        FLOAT* x=IPPF1(ippsMalloc)(d*c);
        FLOAT* y=IPPF1(ippsMalloc)(d*c);
        FLOAT* z=IPPF1(ippsMalloc)(d*c);
        FLOAT* r=IPPF1(ippsMalloc)(c);
    #else
        FLOAT x[d*c];
        FLOAT y[d*c];
        FLOAT z[d*c];
        FLOAT r[c];
    #endif
    DEB(Allocated and collecting positions);
    //#pragma omp parallel for
    for (unsigned int i=0; i<c; ++i) {
        FLOAT* pos=ats[i]->getPos(s);
        for (int j=0; j<d; ++j) {
                x[c*j+i]=pos[3*j];
                y[c*j+i]=pos[3*j+1];
                z[c*j+i]=pos[3*j+2];
        }
        //cout << i << " of " << c << endl;
        r[i]=sqrt(pow(pos[0],2)+pow(pos[1],2)+pow(pos[2],2));
    }
    #ifdef ipp
        //DEB(Calculating MEANS);
        #pragma omp parallel for
        for (int i=0; i<d; ++i) {
                IPPF1(ippsMean)(x+c*i,c,out+3*i);
                IPPF1(ippsMean)(y+c*i,c,out+3*i+1);
                IPPF1(ippsMean)(z+c*i,c,out+3*i+2);
        }
        FLOAT rmean;
        IPPF1(ippsMean)(r,c,&rmean);
        ippsFree(x); ippsFree(y); ippsFree(z); ippsFree(r);
    #else
```

```
        FLOAT rmean=0;
        #pragma omp parallel for
        for (int i=0; i<d; ++i) {
                unsigned int i3=i*3;
                out[i3]=0;
                out[i3+1]=0;
                out[i3+2]=0;
                for (int j=0; j<c; j++) {
                        unsigned int ind=i*c+j;
                        out[i3]+=x[ind];
                        out[i3+1]+=y[ind];
                        out[i3+2]+=z[ind];
                        rmean+=r[j];
                }
                out[i3]/=c;
                out[i3+1]/=c;
                out[i3+2]/=c;
                rmean/=c;
        }
    #endif
    DEB(Done);
    return rmean-1.0;
}

void Particle::getAtomsByType(list<Atom*>& out, vector<atomT*>& types) {
    for (unsigned int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL) continue;
        bool test=0;
        for (int j=0; j<types.size(); ++j) {
                if (atoms[i]->type==types[j]) {
                        test=1;
                        break;
                }
        }
        if (test) {
                out.push_back(atoms[i]);
        }
    }
}

void Particle::getAtomsByType(vector<Atom*>& out, vector<atomT*>& types) {
    for (unsigned int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL) continue;
        bool test=0;
        for (int j=0; j<types.size(); ++j) {
                if (atoms[i]->type==types[j]) {
                        test=1;
                        break;
                }
        }
        if (test) {
                out.push_back(atoms[i]);
        }
    }
}

void Particle::parseByDist(list<Atom*>& in, vector<Atom*>& out, FLOAT* point, FLOAT dist) {
    FLOAT** locs=(FLOAT**)malloc(in.size()*sizeof(FLOAT*));
    unsigned int c=0;
    for (list<Atom*>::iterator i=in.begin(); i!=in.end(); ++i, ++c)
        locs[c]=(*i)->getPos(1);
    c=in.size();
    #ifdef ipp
        FLOAT* rels=IPPF1(ippsMalloc)(3*c);
        FLOAT** pRels=(FLOAT**)malloc(c*sizeof(FLOAT*));
```

```
        for (int i=0; i<c; ++i)
                pRels[i]=rels+3*i;
        int stride2=sizeof(FLOAT);
        IPPF2(ippmSub_vva,L)(point,stride2,locs,0,stride2,pRels,0,stride2,3,c);
        periodicCheck(rels,1,1,c);
        FLOAT* N=IPPF1(ippsMalloc)(c);
        IPPF1(ippmL2Norm_va)(rels,3*stride2,stride2,N,3,c);
        ippsFree(rels); free(pRels);
    #else
        FLOAT* N=(FLOAT*)malloc(c*sizeof(FLOAT));
        //#pragma omp parallel for
        for (int i=0; i<c; i++) {
                FLOAT rel[3];
                rel[0]=point[0]-locs[i][0];
                rel[1]=point[1]-locs[i][1];
                rel[2]=point[2]-locs[i][2];
                periodicCheck(rel,1,1,1);
                for (int j=0; j<3; ++j)
                        N[i]+=pow(rel[j],2);
                N[i]=sqrt(N[i]);
        }
    #endif
    free(locs);
    c=0;
    for (list<Atom*>::iterator i=in.begin(); i!=in.end(); ++i, ++c) {
        if (N[c]<dist)
                out.push_back((*i));
    }
    #ifdef ipp
        ippsFree(N);
    #else
        free(N);
    #endif
}

void Particle::getSilAtoms(vector<Atom*>& silAtoms) {
    atomT* silT=aTypes[8];
    int j=0;
    for (unsigned int i=0; i<atoms.max(); ++i) {
        if (atoms[i]==NULL || atoms[i]->type!=silT)
                continue;
        silAtoms.push_back(atoms[i]);
        ++j;
    }
}

void Particle::printCovMol2(ostream& out, FLOAT* points, bool* cov, int dumps, int N) {
    for (int i=0; i<dumps; ++i) {
        out << "@<TRIPOS>MOLECULE\nCoverage map\n" << 2*N << ' ' << 0 << endl;
        out << "SMALL\nGESTEIGER\nEnergy=0\n\n" << "@<TRIPOS>ATOM\n";
        for (unsigned int j=0; j<N; j++) {
                out << setw (7) << setfill (' ') << left << j+1 << " 1 ";
                if (cov[dumps*j+i]==1)
                        out << " 0.0 0.0 0.0";
                else
                        out << points[3*dumps*j+3*i] << ' ' << points[3*dumps*j+3*i+1] << ' ' <<
    points[3*dumps*j+3*i+2];
                out << " cov 1 LIG1 0.000" << endl;
        }
        for (unsigned int j=0; j<N; j++) {
                out << setw (7) << setfill (' ') << left << N+j+1 << " 2 ";
                if (cov[dumps*j+i]==0)
                        out << " 0.0 0.0 0.0";
                else
                        out << points[3*dumps*j+3*i] << ' ' << points[3*dumps*j+3*i+1] << ' ' <<
```

```
    points[3*dumps*j+3*i+2];
                out << " exp 2 LIG1 0.000" << endl;
        }
        out << "@<TRIPOS>BOND\n";
    }
}

void Particle::calcDirections(ostream& out, FLOAT* centers) {
    //Needs to follow general basis by which dihedral works
    int l[molecules.max()];
    FLOAT* dirs[molecules.max()];
    #pragma omp parallel for
    for (unsigned int i=0; i<molecules.max(); ++i) {
        if (molecules[i]==NULL) { l[i]=0; continue; }
        l[i]=molecules[i]->calcChainDir(dirs[i],centers,1,dumps);
    }
    for (int i=0; i<dumps; ++i) {
        out << "TIMESTEP: " << timesteps[i] << endl;
        unsigned int k=1;
        for (unsigned int j=0; j<molecules.max(); ++j) {
                if (l[j]==0) continue;
                out << setw(8) << left << k;
                k++;
                for (int f=0; f<l[j]; ++f) {
                        out << setw(15) << setprecision(10) << left <<  dirs[j][f*dumps+i];
                }
                out << endl;
        }
    }
    for (unsigned int i=0; i<molecules.max(); ++i) {
        #ifdef ipp
                if (l[i]!=0) ippsFree(dirs[i]);
        #else
                if (l[i]!=0) free(dirs[i]);
        #endif
    }
}
```

Particle Range-of-Motion Calculation **ParticleROM.cpp**

```
#include "inc.h"

void Particle::ROM(int interval, ostream& out) {
  //First need to collect molecules, and find ends and anchors
  vector<Atom*> ancs,ends;
  Atom *end, *anc, *a;
  for (int i = 0; i<molecules.size(); ++i) {
    end = NULL; anc = NULL;
    if (molecules[i]==NULL) continue;
    if (molecules[i]->atoms.size()>100) continue;
    list<Atom*>::iterator j = molecules[i]->atoms.begin();
    while (anc == NULL || end == NULL) {
      a = *j;
      if (a->type == aTypes[8])
        anc = a;
      else if (a->type == aTypes[11])
        end = a;
      ++j;
    }
    ancs.push_back(anc);
    ends.push_back(end);
  }
  FLOAT** aPos = new FLOAT*[ancs.size()];
  FLOAT** ePos = new FLOAT*[ends.size()];
```

```
    int boxes = dumps/interval;
    int offset = dumps%interval;
    FLOAT *rom = new FLOAT[boxes];
  for (int i = 0; i<ancs.size(); ++i) {
    aPos[i] = ancs[i]->getPos(offset);
    ePos[i] = ends[i]->getPos(offset);
  }
  out << interval << '\t' << ends.size() << '\t' << boxes << '\t' << offset << endl;
  double *rdata = new double[3*ends.size()*interval*boxes]; //used to make 3 dimensional data set
  double **ri = new double*[ends.size()];
  for (int i = 0; i<ends.size(); ++i) {
      ri[i] = rdata + i*(3*interval*boxes); //full stride is 3*all timesteps considered
  }
  int stride2 = sizeof(FLOAT);
  IPPF2(ippmSub_vava,L)(ePos,0,stride2,aPos,0,stride2,ri,0,stride2,3*interval*boxes,ancs.size());
  for (int i = 0; i<ends.size(); ++i) {
    for (int j = 0; j < interval*boxes; ++j) {
      minimumImage(ri[i]+3*j,offset+j);
    }
  }
  double rTmp,dx,dy,dz;
  int x,y,z,i,n,m;
  #pragma omp parallel for private(rTmp,dx,dy,dz,x,y,z,i,n,m)
  for (int b = 0; b < boxes; ++b) {
    rom[b] = 0;
    x = 3*b*interval;
    y = 3*b*interval+1;
    z = 3*b*interval+2;
    for (i = 0; i<ancs.size(); ++i) {
      rTmp = 0;
      for (n = 0; n < interval; ++n) {
        for (m = n+1; m < interval; ++m) {
          dx = ri[i][x+3*n]-ri[i][x+3*m];
          dy = ri[i][y+3*n]-ri[i][y+3*m];
          dz = ri[i][z+3*n]-ri[i][z+3*m];
          rTmp += sqrt(dx*dx+dy*dy+dz*dz);
        }
      }
      rTmp *= 2.0/double(interval*(interval-1));
      rom[b] += rTmp;
    }
    rom[b] /= ancs.size();
  }
  for (int b = 0; b < boxes; ++b) {
        out << timesteps[offset+b*interval+interval/2] << " " << offset+b*interval+interval/2 <<
    " " << rom[b] << endl;
    }
  delete [] aPos; delete [] ePos; delete [] ri; delete [] rdata; delete [] rom;
}
```

Particle PBC Replication Functions — **ParticleRepl.cpp**

```
#ifndef ipp
    #include "bMathLibs.h"
#endif
#include "inc.h"

void Particle::replicate(int nx, int ny, int nz) {
    cout << "Replicating x dir " << nx << " times" << endl;
    singDimRep(0,nx);
    cout << "Replicating y dir " << ny << " times" << endl;
    singDimRep(1,ny);
    cout << "Replicating z dir " << nz << " times" << endl;
    singDimRep(2,nz);
```

```
    cout << "Verifying Connectivity" << endl;
    verifyCon(0);
    cout << "Replication done" << endl;
}

void Particle::singDimRep(int dimen, int n) {
    molecules.replicate(dimen,n);
    dim[2*dimen+1]=dim[2*dimen]+(n+1)*perDim[dimen];
    perDim[dimen]=(n+1)*perDim[dimen];
    compressAll();
}

void MolecList::replicate(int dim, int n) {
    compress();
    if (n==0)
        return;
    int iniC=m;
    for (int i=0; i<iniC; ++i) {
        if (list[i]==NULL)
                continue;
        Molecule* ms[n];
        if (list[i]->atoms.size()>100) {
                for (int j=0; j<n; ++j) {
                        ms[j]=list[i];
                }
        } else {
                for (int j=0; j<n; ++j) {
                        ms[j]=new Molecule();
                        addElem(ms[j]);
                }
        }
        list[i]->replicate(dim,n,ms);
    }
}

void MolecList::replicate(int dim, map<unsigned int,vector<Atom*> >& images) {
    compress();
    int iniC=c;
    int n=images.begin()->second.size();
    for (unsigned int i=0; i<iniC; ++i) {
        //First case, molecule is huge, assume it spans all cells and assign all image atoms to
    this one
        if (list[i]->atoms.size()>100) {
                vector<Atom*> newAts;
                for (std::list<Atom*>::iterator j=list[i]->atoms.begin(); j!=list[i]-
    >atoms.end(); ++j) {
                        for (int k=0; k<n; ++k) {
                                newAts.push_back(images[(*j)->lIndex][k]);
                        }
                }
                for (int k=0; k<newAts.size(); k++) {
                        list[i]->addAtom(newAts[k]);
                }
                continue;
        }
        //Next case assumes that it is small, and may or may not span cells
        set<Atom*> l,r;
        bool bnd=list[i]->leftRight(dim,l,r);
        if (bnd==0) {
                //Doesn't span cells
                for (int j=0; j<n; ++j) {
                        Molecule* m=new Molecule();
                        for (std::list<Atom*>::iterator k=list[i]->atoms.begin(); k!=list[i]-
    >atoms.end(); ++k) {
                                m->addAtom(images[(*k)->lIndex][j]);
```

```
                        }
                        addElem(m);
                }
        } else {
                //Spans cells, first do right side of all new cells except last
                for (int j=0; j<n-1; ++j) {
                        Molecule* m=new Molecule();
                        for (set<Atom*>::iterator k=l.begin(); k!=l.end(); ++k) {
                                m->addAtom(images[(*k)->lIndex][j+1]);
                        }
                        for (set<Atom*>::iterator k=r.begin(); k!=r.end(); ++k) {
                                m->addAtom(images[(*k)->lIndex][j]);
                        }
                        addElem(m);
                }
                Molecule* m1=new Molecule(); //This mol spans ini cell and next cell
                for (set<Atom*>::iterator k=l.begin(); k!=l.end(); ++k) {
                        //left atoms are moved out of this molecule and into new one
                        //then replaced by atoms from the next cell over
                        list[i]->remAtom(*k);
                        list[i]->addAtom(images[(*k)->lIndex][0]);
                        m1->addAtom(*k);
                }
                for (set<Atom*>::iterator k=r.begin(); k!=r.end(); ++k) {
                        //right atoms cell n-1 are attached to m1
                        m1->addAtom(images[(*k)->lIndex][n-1]);
                }
                addElem(m1);
        }
    }
}

void Molecule::replicate(int dim, int n, Molecule* ms[]) {
    set<unsigned int> bs, as, ds, is;
    map<unsigned int, unsigned int* > ax;
    set<Atom*> left, right;
    bool isBnd=leftRight(dim,left,right);
    for (std::list<Atom*>::iterator i=atoms.begin(); i!=atoms.end(); ++i) {
        (*i)->collectConn(bs,as,ds,is);
    }
    int I=ax.size();
    parent->parent->atoms.replicate(dim,n,*this,ms,left,right,ax);
    parent->parent->bonds.replicate(dim,n,bs,ax);
    parent->parent->angles.replicate(dim,n,as,ax);
    parent->parent->dihedrals.replicate(dim,n,ds,ax);
    parent->parent->impropers.replicate(dim,n,is,ax);
    int k=3540;
    for (map<unsigned int,unsigned int*>::iterator i=ax.begin(); i!=ax.end(); ++i) {
        free(i->second);
    }
```

Particle Combining Functions **ParticleCombine.cpp**

```
#ifndef ipp
    #include "bMathLibs.h"
#endif
#include "inc.h"

void Particle::translate(FLOAT t[]) {
    atoms.translate(t);
    dim[0]+=t[0];
    dim[1]+=t[0];
    dim[2]+=t[1];
    dim[3]+=t[1];
```

```
    dim[4]+=t[2];
    dim[5]+=t[2];
}

void Particle::translatePer(FLOAT t[]) {
    atoms.translatePer(t);
}



void Particle::combineParticles(Particle &P) {
    for (int i=0; i<3; ++i) {
        if (periodic[i]) continue;
        if (P.dim[2*i]<dim[2*i])
                dim[2*i]=P.dim[2*i];
        if (P.dim[2*i+1]>dim[2*i+1])
                dim[2*i+1]=P.dim[2*i+1];
    }
    prepCombine(P);
    transAll(P);
    aTypes.addDummyType();
}

void Particle::transAll(Particle &P) {
    map<unsigned int,Atom*> ax;
    atoms.transAll(P,ax);
    //DEB(Atoms Transferred)
    #pragma omp sections
    {
        #pragma omp section
        {
                bonds.transAll(P,ax);
                //DEB(B)
        }
        #pragma omp section
        {
                angles.transAll(P,ax);
                //DEB(A)
        }
        #pragma omp section
        {
                dihedrals.transAll(P,ax);
                //DEB(D)
        }
        #pragma omp section
        {
                impropers.transAll(P,ax);

        }
        #pragma omp section
        {
                molecules.transAll(P,ax);
        }
    }
}

void Particle::cleave(FLOAT c,int d) {
    atoms.cleave(c,d);
}

void Particle::cleaveCyl(FLOAT c, int d) {
    atoms.cleaveCyl(c,d);
}

void Particle::cleave(FLOAT *c) {
```

```
  atoms.cleave(c);
}
```

APPENDIX C

ROUTINES FOR GENERATION AND ANALYSIS OF PRESSURE AND STRAIN DISTRIBUTION MAPS

C.1 Modified LAMMPS Computes for Contact Analysis

Computes were developed to provide the necessary data for analysis of spatially resolved pressure distributions.

C.1.1 Interaction Computations

Compute 'atom/groups_red' was developed to compute the pairwise interaction energies and forces between sets of atoms in different groups. "atom" indicates that the data is provided on a per atom basis, "groups" indicates that the interactions can be between multiple groups, and 'red' means reduce the data as well, producing summed values of the interaction energies and forces. A compute group is specified in LAMMPS, and subsequent groups are the interaction groups. For a 4 component system (i.e. particle-film-film-particle), one compute was specified for the first particle with interacting groups "film-film-particle", next compute would be first film with interacting groups "film-particle" and so on. The compute stores per atom data for the compute group and the interacting group. This compute was derived from the LAMMPS compute group/group which provides a global measurement of the interaction energy and forces between two groups, extended to support multiple groups and per atom data.

atom/groups_red Compute Declaration **compute_atom_groups_red.h**

```
#ifdef COMPUTE_CLASS

ComputeStyle(atom/groups_red,ComputeAtomGroupsRed)

#else
```

```
#ifndef LMP_COMPUTE_ATOM_GROUPS_RED_H
#define LMP_COMPUTE_ATOM_GROUPS_RED_H

#include "compute.h"

namespace LAMMPS_NS {

class ComputeAtomGroupsRed : public Compute {
 public:
  ComputeAtomGroupsRed(class LAMMPS *, int, char **);
  ~ComputeAtomGroupsRed();
  void init();
  void init_list(int, class NeighList *);
  void compute_peratom();
  void compute_vector();
  double memory_usage();
  int pack_reverse_comm(int,int,double*);
  void unpack_reverse_comm(int, int *, double*);

 private:

  char **oGroups;
  int nJgroups;
  int *jgroups,*jgroupbits, jGrpCheck;
  bigint nmax;
  double **cutsq;
  class Pair *pair;
  class NeighList *list;

  void interact();
};

}

#endif
#endif

/* ERROR/WARNING messages:

E: Illegal ... command

Self-explanatory.  Check the input script syntax and compare to the
documentation for the command.  You can use -echo screen as a
command-line option when running LAMMPS to see the offending line.

E: Compute group/group group ID does not exist

Self-explanatory.

E: No pair style defined for compute group/group

Cannot calculate group interactions without a pair style defined.

E: Pair style does not support compute group/group

The pair_style does not have a single() function, so it cannot be
invokded by the compute group/group command.

*/
```

Atom/groups_red Compute Class Definition **compute_atom_groups_red.cpp**

```
#include "mpi.h"
#include "string.h"
```

```
#include "compute_atom_groups_red.h"
#include "atom.h"
#include "update.h"
#include "comm.h"
#include "force.h"
#include "pair.h"
#include "memory.h"
#include "neighbor.h"
#include "neigh_request.h"
#include "neigh_list.h"
#include "group.h"
#include "domain.h"
#include "error.h"

using namespace LAMMPS_NS;

/* ---------------------------------------------------------------------- */

ComputeAtomGroupsRed::ComputeAtomGroupsRed(LAMMPS *lmp, int narg, char **arg) :
  Compute(lmp, narg, arg)
{
  if (narg < 4) error->all(FLERR,"Illegal compute atom/groups command");

  //My modifications
  nJgroups = narg-3;
  oGroups = new char*[nJgroups];
  jgroups = new int[nJgroups];
  jgroupbits = new int[nJgroups];

  peratom_flag=1;
  size_peratom_cols=4*(nJgroups+1);
  array_flag=1;
  vector_flag=1;

  size_vector = size_peratom_cols;
  vector = new double[size_vector];
  comm_reverse = size_vector;

  //End My Modifications
  for (int i=0; i<nJgroups; ++i) {
      int n = strlen(arg[3+i]) + 1;
      oGroups[i] = new char[n];
      strcpy(oGroups[i],arg[3+i]);

      jgroups[i] = group->find(oGroups[i]);
      if (jgroups[i] == -1)
        error->all(FLERR,"Compute atom/group group ID does not exist");
      jgroupbits[i] = group->bitmask[jgroups[i]];
     if (screen) fprintf(screen,"%s in atom/groups compute\n",oGroups[i]);
  }

  jGrpCheck = 0;
  for (int k = 0; k < nJgroups; ++k)
      jGrpCheck|=jgroupbits[k];
  nmax=0;
  array_atom=NULL;
  array=NULL;


}

/* ---------------------------------------------------------------------- */

ComputeAtomGroupsRed::~ComputeAtomGroupsRed()
{
```

```
  delete [] array;
  for (int i=0; i<nJgroups; ++i) {
      delete [] oGroups[i];
  }
  delete [] oGroups;
  delete [] jgroups;
  delete [] jgroupbits;
  delete [] vector;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsRed::init()
{
  if (force->pair == NULL)
    error->all(FLERR,"No pair style defined for compute atom/group");

  // if non-hybrid, then error if single_enable = 0
  // if hybrid, let hybrid determine if sub-style sets single_enable = 0

  if (force->pair_match("hybrid",0) == NULL && force->pair->single_enable == 0)
    error->all(FLERR,"Pair style does not support compute atom/group");
  if (!(force->newton_pair))
    error->all(FLERR,"Newton pairing must be turned on for compute atom/groups");

  pair = force->pair;
  cutsq = force->pair->cutsq;

  // recheck that group 2 has not been deleted

  for (int i = 0; i<nJgroups; ++i) {
      jgroups[i] = group->find(oGroups[i]);
      if (jgroups[i] == -1)
        error->all(FLERR,"Compute atom/group group ID does not exist");
      jgroupbits[i] = group->bitmask[jgroups[i]];
  }
  jGrpCheck = 0;
  for (int k = 0; k < nJgroups; ++k)
      jGrpCheck|=jgroupbits[k];

  // need an occasional half neighbor list

  int irequest = neighbor->request((void *) this);
  neighbor->requests[irequest]->pair = 0;
  neighbor->requests[irequest]->compute = 1;
  neighbor->requests[irequest]->occasional = 1;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsRed::init_list(int id, NeighList *ptr)
{
  list = ptr;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsRed::compute_peratom()
{
  if (atom->nmax > nmax) {
    nmax = atom->nmax;
    memory->destroy(array);
    memory->create(array,nmax,size_peratom_cols,"atom/groups:peratom");
    array_atom = array;
  }
```

```
  invoked_peratom = update->ntimestep;
  invoked_vector = update->ntimestep;

  interact();
}

void ComputeAtomGroupsRed::compute_vector()
{
  if (atom->nmax > nmax) {
    nmax = atom->nmax;
    memory->destroy(array);
    memory->create(array,nmax,size_peratom_cols,"atom/groups:peratom");
    array_atom = array;
  }
  invoked_peratom = update->ntimestep;
  invoked_vector = update->ntimestep;

  interact();
}

/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
void ComputeAtomGroupsRed::interact()
{
  //outside needed data
  double **x = atom->x;
  int *type = atom->type;
  int *mask = atom->mask;
  int nlocal = atom->nlocal;
  double *special_coul = force->special_coul;
  double *special_lj = force->special_lj;
  int newton_pair = force->newton_pair;
  neighbor->build_one(list->index);

  //list data
  int inum = list->inum;
  int *ilist = list->ilist;
  int *numneigh = list->numneigh;
  int **firstneigh = list->firstneigh;

  //temperorary constants
  int *jlist;
  int i, j, k, m, k4, atom1, atom2, a[2], t[2], jnum;
  double factor_lj, factor_coul, rsq, epair, fpair, del[3];

  //storage
  double *one = new double[size_vector];
  double *all = new double[size_vector];
  for (i = 0; i<size_vector; ++i)
      one[i]=0;
  for (k = 0; k < nmax; ++k) {
      for (i = 0; i < size_vector; ++i)
          array[k][i] = 0;
  }
  int allgroups = groupbit | jGrpCheck;
  for (i = 0; i < inum; ++i) {
      atom1 = ilist[i];
    if (!(mask[atom1]&allgroups)) continue;
    jlist = firstneigh[atom1];
    jnum = numneigh[atom1];
    for (j = 0; j < jnum; ++j) {
        atom2 = jlist[j];
        factor_lj = special_lj[sbmask(atom2)];
        factor_coul = special_lj[sbmask(atom2)];
```

```
      atom2 &= NEIGHMASK;         //strip neighbor flags off of atom2
      if (!(mask[atom2]&allgroups)) continue; //at this point, both atoms could be in main
  group or jgroups, make sure both are represetned
      if (mask[atom1]&groupbit && mask[atom2]&jGrpCheck) {
          a[0] = atom1;
          a[1] = atom2;
      } else if (mask[atom2]&groupbit && mask[atom1]&jGrpCheck) {
          a[0] = atom2;
          a[1] = atom1;
      } else
          continue; //neither atom is in main group
      t[0] = type[a[0]];
      t[1] = type[a[1]];
      for (m = 0; m < 3; ++m)
          del[m] = x[a[0]][m] - x[a[1]][m];
      domain->minimum_image(del);
      rsq = del[0]*del[0] + del[1]*del[1] + del[2]*del[2];
      epair = pair->single(a[0],a[1],t[0],t[1],rsq,factor_coul,factor_lj,fpair)*0.5;
      for (m = 0; m < 3; ++m)
          del[m] *= fpair;
      for (k = 0; k < nJgroups; ++k) {
          k4 = 4*k;
          if (mask[a[1]] & jgroupbits[k]) {
              if (jgroupbits[k] & groupbit) {
                  array[a[0]][k4] += epair;
                  array[a[1]][k4] += epair;
                  one[k4]         += 2.0*epair;
                  for (m = 0; m < 3; ++m) {
                      array[a[0]][k4+m+1] += del[m];
                      array[a[1]][k4+m+1] -= del[m];
                  }
              } else {
                  array[a[0]][k4] += epair;
                  one[k4]         += epair;
                  for (m = 0; m < 3; ++m) {
                      array[a[0]][k4+m+1] += del[m];
                      one[k4+m+1]         += del[m];
                  }
              }
          }
      }
      k4 = nJgroups*4;
      if (jGrpCheck & groupbit && mask[a[1]] & groupbit) {
          array[a[0]][k4] += epair;
          array[a[1]][k4] += epair;
          one[k4]         += 2.0*epair;
          for (m = 0; m < 3; ++m) {
              array[a[0]][k4+m+1] += del[m];
              array[a[1]][k4+m+1] -= del[m];
          }
      } else {
          array[a[0]][k4] += epair;
          one[k4]         += epair;
          for (m = 0; m < 3; ++m) {
              array[a[0]][k4+m+1] += del[m];
              one[k4+m+1]         += del[m];
          }
      }
    }
  }

  comm->reverse_comm_compute(this);
  //need to aggregate per atom data per node...
  MPI_Allreduce(one,all,size_vector,MPI_DOUBLE,MPI_SUM,world);
  for (i = 0; i < size_vector; ++i)
```

```
      vector[i] = all[i];
  delete [] one;
  delete [] all;
}

/* ---------------------------------------------------------------------- */

int ComputeAtomGroupsRed::pack_reverse_comm(int n, int first, double *buf)
{
  int i,m,k,last;

  m = 0;
  last = first + n;
  for (i = first; i < last; i++) {
      for (k = 0; k < size_vector; ++k)
          buf[m++] = array[i][k];
  }
  return size_vector;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsRed::unpack_reverse_comm(int n, int *llist, double *buf)
{
  int i,j,k,m;

  m = 0;
  for (i = 0; i < n; i++) {
    j = llist[i];
    for (k = 0; k < size_vector; ++k)
        array[j][k] += buf[m++];
  }
}

double ComputeAtomGroupsRed::memory_usage()
{
  double bytes = nmax * size_vector * sizeof(double) + size_vector * sizeof(double);
  return bytes;
}
```

Compute atom/group_conn was developed to consider the non-pairwise interaction forces between two atomic groups in the LAMMPS simulation. This is specifically designed to collect the extra terms of interaction between the particle surface and the bound film, where bonding and angular terms in addition to pairwise interactions. The interaction forces and energies were divided half-and-half between the compute group and the interaction group.

Compute atom/group_conn Declaraction **compute_atom_group_conn.h**

```
#ifdef COMPUTE_CLASS

ComputeStyle(atom/group_conn,ComputeAtomGroupsConn)
```

```
#else

#ifndef LMP_COMPUTE_ATOM_GROUPS_CONN_H
#define LMP_COMPUTE_ATOM_GROUPS_CONN_H

#include "compute.h"
#include "bond.h"
#include "angle.h"

namespace LAMMPS_NS {

class ComputeAtomGroupsConn : public Compute {
 public:
  ComputeAtomGroupsConn(class LAMMPS *, int, char **);
  ~ComputeAtomGroupsConn();
  void init();
  void init_list(int, class NeighList *);
  void compute_peratom();
  void compute_vector();
  double memory_usage();
  int pack_reverse_comm(int,int,double*);
  void unpack_reverse_comm(int, int *, double*);

 private:

  char *jGroupName;
  int nJgroups;
  int jgroup,jgroupbit;
  bigint nmax;
  Angle *angle;
  Bond *bond;

  void interact();
};

}

#endif
#endif
```

Compute atom/group_conn Definition **compute_atom_group_conn.cpp**

```
#include "mpi.h"
#include "string.h"
#include "compute_atom_group_conn.h"
#include "atom.h"
#include "update.h"
#include "comm.h"
#include "force.h"
#include "pair.h"
#include "memory.h"
#include "neighbor.h"
#include "neigh_request.h"
#include "neigh_list.h"
#include "group.h"
#include "domain.h"
#include "error.h"

using namespace LAMMPS_NS;

/* ---------------------------------------------------------------------- */

ComputeAtomGroupsConn::ComputeAtomGroupsConn(LAMMPS *lmp, int narg, char **arg) :
  Compute(lmp, narg, arg)
```

```
{
  if (narg != 4) error->all(FLERR,"Illegal compute atom/groups command");

  //My modifications

  peratom_flag=1;
  array_flag=1;
  vector_flag=1;

  size_peratom_cols=4;

  size_vector = size_peratom_cols;
  vector = new double[size_vector];
  comm_reverse = size_vector;

  //End My Modifications
  int n = strlen(arg[3]);
  jGroupName = new char[n];
  strcpy(jGroupName,arg[3]);
  jgroup = group->find(jGroupName);
  if (jgroup == -1)
      error->all(FLERR,"Compute atom/group_conn group ID does not exist");
  jgroupbit = group->bitmask[jgroup];

  nmax=0;
  array_atom=NULL;
  array=NULL;
}

/* ---------------------------------------------------------------------- */

ComputeAtomGroupsConn::~ComputeAtomGroupsConn()
{
  delete [] array;
  delete [] jGroupName;
  delete [] vector;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsConn::init()
{
  if (force->bond == NULL)
    error->all(FLERR,"No bond style defined for compute atom/group_conn");
  if (force->angle == NULL)
      error->all(FLERR,"No angle style defined for compute atom/group_conn");

  // if non-hybrid, then error if single_enable = 0
  // if hybrid, let hybrid determine if sub-style sets single_enable = 0

  if (!(force->newton_bond))
    error->all(FLERR,"Newton bonding must be turned on for compute atom/group_conn");

  angle = force->angle;
  bond = force->bond;
  // recheck that group 2 has not been deleted
  jgroup = group->find(jGroupName);
  if (jgroup==-1)
      error->all(FLERR,"Compute atom/group_conn group ID does not exist");
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsConn::init_list(int id, NeighList *ptr)
{
```

```
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsConn::compute_peratom()
{
  if (atom->nmax > nmax) {
    nmax = atom->nmax;
    memory->destroy(array);
    memory->create(array,nmax,size_peratom_cols,"atom/groups:peratom");
    array_atom = array;
  }
  invoked_peratom = update->ntimestep;
  invoked_vector = update->ntimestep;

  interact();
}

void ComputeAtomGroupsConn::compute_vector()
{
  if (atom->nmax > nmax) {
    nmax = atom->nmax;
    memory->destroy(array);
    memory->create(array,nmax,size_peratom_cols,"atom/groups:peratom");
    array_atom = array;
  }
  invoked_peratom = update->ntimestep;
  invoked_vector = update->ntimestep;

  interact();
}

/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
void ComputeAtomGroupsConn::interact()
{

    //Outside values needed pertaining to bonds
  int *num_bond = atom->num_bond;
  int **bond_atom = atom->bond_atom;
  int **bond_type = atom->bond_type;

  //outside values needed pertaining to atoms
  double **x = atom->x;
  int *mask = atom->mask;
  int *tag = atom->tag;
  int nlocal = atom->nlocal;
  int newton_bond = force->newton_bond;

  //variables needed in computation
  int atom1, atom2, atom3, i, j, k, k4,type;
  int a[3],m[3];
  double del[3],del2[3], ebond, fbond, rsq, rsq2, r[2];

  //storage values
  double *one = new double[size_vector];
  double *all = new double[size_vector];
  for (int i = 0; i<size_vector; ++i)
      one[i]=0;
  for (k = 0; k < nmax; ++k) {
      for (i = 0; i < size_vector; ++i)
          array[k][i] = 0;
  }
```

```
int bothgroups = groupbit | jgroupbit;

for (atom1 = 0; atom1 < nlocal; ++atom1) {
    if (!(bothgroups&mask[atom1])) continue;
    for (i = 0; i < num_bond[atom1]; ++i) {
        atom2 = atom->map(bond_atom[atom1][i]);
        if (atom2 < 0 || !(mask[atom2]&bothgroups)) continue;
        if (bond_type[atom1][i]>0)
            type = bond_type[atom1][i];
        else
            type = -bond_type[atom1][i];
        if (type == 0) continue;
        if (mask[atom1]&groupbit && mask[atom2]&jgroupbit) {
            a[0] = atom1;
            a[1] = atom2;
        } else if (mask[atom1]&jgroupbit && mask[atom2]&groupbit) {
            a[0] = atom2;
            a[1] = atom1;
        } else continue;
        for (j = 0; j < 3; ++j)
            del[j] = x[a[0]][j] - x[a[1]][j];
        domain->minimum_image(del);
        rsq = del[0]*del[0] + del[1]*del[1] + del[2]*del[2];
        ebond = bond->single(type,rsq,a[0],a[1],fbond)*0.5;
        for (j = 0; j < 3; ++j)
            del[j]*=fbond;
        array[a[0]][0]    += ebond;
        one[0]            += ebond;
        if (jgroupbit & groupbit) {
            array[a[1]][0]    += ebond;
            one[0]            += ebond;
            for (j = 0; j < 3; ++j) {
                array[a[0]][j+1] += del[j];
                array[a[1]][j+1] -= del[j];
            }
        } else {
            for (j = 0; j < 3; ++j) {
                array[a[0]][j+1] += del[j];
                one[j+1]         += del[j];
            }
        }
    }
}

int *num_angle = atom->num_angle;
int **angle_atom1 = atom->angle_atom1;
int **angle_atom2 = atom->angle_atom2;
int **angle_atom3 = atom->angle_atom3;
int **angle_type = atom->angle_type;
int allMask;
double f1[3], f3[3], eangle;
int **anglelist = neighbor->anglelist;
int nanglelist = neighbor->nanglelist;

for (atom2 = 0; atom2 < nlocal; atom2++) {
    if (!(mask[atom2] & bothgroups)) continue;
    for (i = 0; i < num_angle[atom2]; ++i) {
        if (tag[atom2] != angle_atom2[atom2][i]) continue;
        if (angle_type[atom2][i]>0)
            type = angle_type[atom2][i];
        else
            type = -angle_type[atom2][i];
        if (type == 0) continue;
        atom1 = atom->map(angle_atom1[atom2][i]);
        if (atom1 < 0 || !(mask[atom1] & bothgroups)) continue;
```

```
      atom3 = atom->map(angle_atom3[atom2][i]);
      if (atom3 < 0 || !(mask[atom3] & bothgroups)) continue;
      allMask = mask[atom1] | mask[atom2] | mask[atom3];
      if (allMask & groupbit && allMask & jgroupbit) {  // at least one atom in group and
jgroup
        a[0] = atom1;
        a[1] = atom2;
        a[2] = atom3;
      } else continue; //both groups are not represented, move on
      m[0] = mask[a[0]];
      m[1] = mask[a[1]];
      m[2] = mask[a[2]];
      for (j = 0; j < 3; ++j) {
          del[j] = x[a[0]][j] - x[a[1]][j];
          del2[j] = x[a[2]][j] - x[a[1]][j];
      }
      domain->minimum_image(del);
      domain->minimum_image(del2);
      eangle = angle->single(type,del,del2,f1,f3)/3.0;
      if (groupbit & jgroupbit) {
          //jg-jg-jg, all initial screens passed means this does not need to be screened
        array[a[0]][0]    += eangle;
        array[a[1]][0]    += eangle;
        array[a[2]][0]    += eangle;
        one[0]            += eangle*3.0;
        for (j = 0; j < 3; ++j) {
            array[a[0]][j+1] += f1[j];
            array[a[1]][j+2] -= f1[j] + f3[j];
            array[a[2]][j+3] += f3[j];
        }
      } else {
          if (m[0] & groupbit) { //g-x-x
              if (m[1] & groupbit) { //g-g-x
                  if (m[2] & jgroupbit) { //g-g-j
                      array[a[1]][0] += eangle;
                      one[0] += eangle;
                      for (j = 0; j < 3; ++j) {
                          array[a[1]][j+1]  -= f3[j];
                          one[j+1]          -= f3[j];
                      }
                  }
              } else if (m[1] & jgroupbit) { //g-j-x
                  if (m[2] & groupbit) { //g-j-g
                      array[a[0]][0] += eangle*0.5;
                      array[a[2]][0] += eangle*0.5;
                      one[0] += eangle;
                      for (j = 0; j < 3; ++j) {
                          array[a[0]][j+1]  += f1[j];
                          array[a[2]][j+1]  += f3[j];
                          one[j+1]          += f1[j]+f3[j];
                      }
                  } else if (m[2] & jgroupbit) { //g-j-j
                      array[a[0]][0] += eangle;
                      one[0] += eangle;
                      for (j = 0; j < 3; ++j) {
                          array[a[0]][j+1]  += f1[j];
                          one[j+1]          += f1[j];
                      }
                  }
              }
          } else { //j-x-x
              if (m[1] & groupbit) { //j-g-x
                  if (m[2] & groupbit) { //j-g-g
                      array[a[1]][0]    += eangle;
                      one[0]            += eangle;
```

```
                        for (j = 0; j < 3; ++j) {
                            array[a[1]][j+1]  -= f1[j];
                            one[j+1]          -= f1[j];
                        }
                    } else { //j-g-j
                        array[a[1]][0] += eangle;
                        one[0] += eangle;
                        for (j = 0; j < 3; ++j) {
                            array[a[1]][0] -= f1[j] + f3[j];
                            one[j+1]       -= f1[j] + f3[j];
                        }
                    }
                } else { //j-j-g
                    array[a[2]][0] += eangle;
                    one[0] += eangle;
                    for (j = 0; j < 3; ++j) {
                        array[a[2]][j+1] += f3[j];
                        one[j+1]          += f3[j];
                    }
                }
            }
        }
    }
  }
  comm->reverse_comm_compute(this);
  //need to aggregate per atom data per node...
  MPI_Allreduce(one,all,size_vector,MPI_DOUBLE,MPI_SUM,world);
  for (i = 0; i < size_vector; ++i)
      vector[i] = all[i];
  delete [] one;
  delete [] all;
}

/* ---------------------------------------------------------------------- */

int ComputeAtomGroupsConn::pack_reverse_comm(int n, int first, double *buf)
{
  int i,m,k,last;

  m = 0;
  last = first + n;
  for (i = first; i < last; i++) {
      for (k = 0; k < size_vector; ++k)
          buf[m++] = array[i][k];
  }
  return size_vector;
}

/* ---------------------------------------------------------------------- */

void ComputeAtomGroupsConn::unpack_reverse_comm(int n, int *llist, double *buf)
{
  int i,j,k,m;

  m = 0;
  for (i = 0; i < n; i++) {
    j = llist[i];
    for (k = 0; k < size_vector; ++k)
        array[j][k] += buf[m++];
  }
}

double ComputeAtomGroupsConn::memory_usage()
{
  double bytes = nmax * size_vector * sizeof(double) + size_vector * sizeof(double);
```

```
  return bytes;
}
```

C.2 Pressure Map Generation and Fitting Functions

The atomic data generated by LAMMPS was used to prepare pressure and strain energy maps. These were generated by combining time averaged atomic position and interaction property data. "cudaConvForce" is designed to perform the convolution of these data sets (described by Equation 4.3), employing the CUDA library to dramatically speed up the convolution process by parallelizing the generation of the two dimensional data set.

Main Program Routine **main.cpp**

```
#include <iostream>
#include <fstream>
#include <string.h>
#include <map>
#include <cuda_runtime.h>
#include <float.h>

using namespace std;

void getFWHM(char input[], map<int,float> &fwhm);
float* getPos(char input[], int &rows, int &steps, int* &timesteps, int* &types, map<int,int>*
    &ind2row);
float* getAtomData(char input[], int steps, int &rows, int &cols, char** &headers, int* &ids);
void sortAtomData(int *ids, float* &pos, int* &types, int steps, int dRows, map<int,int>*
    ind2row);
float* processData(float *data, float *pos, int dim[], int *types, map<int,float> &fwhm, int
    steps, int cols, int rows, float multipliers[], int res, float min, float max);
void setDataRange(float *pos,int rows, int dims[],float &min, float &max, float width);
void collectData(char input[],float* &pos, float* &data, int &steps, int &rows, int &cols, int*
    &timesteps, char** &headers);

int main(int argc, char* argv[]) {
    //first input is the input file
    //input count should ultimately then be 1+1+1+1+numCols*2, a php script can handle the actual
    running of this command
    char inpName[512];
    char baseName[512];
    strcpy(inpName,argv[1]);
    int dim[2],res;
    if (strstr(inpName,"Flat")) {
        dim[0] = 0;
        dim[1] = 1;
    } else if (strstr(inpName,"NX")) {
        dim[0] = 1;
        dim[1] = 2;
    } else if (strstr(inpName,"NY")) {
        dim[0] = 0;
        dim[1] = 2;
```

```
    } else if (strstr(inpName,"NZ")) {
        dim[0] = 0;
        dim[1] = 1;
    }
    cout << "dim0 " << dim[0] << " dim1 " << dim[1] << endl;
    int argPos;
    map<int,float> fwhm;
    float *pos,*data,width,*multipliers;
    int *timesteps, *types;
    char **headers, **units;
    int steps,cols,dRows,startSave;
    float min,max;
    if (strstr(inpName,"aData")) {
        startSave = 1;
        char posName[512];
        char dataName[512];
        strcpy(posName,argv[2]);
        strcpy(dataName,argv[3]);
        getFWHM(dataName,fwhm);
        int posRows;
        map<int,int> *ind2Row;
        pos = getPos(posName,posRows,steps,timesteps,types,ind2Row);
        int *ids;
        data = getAtomData(inpName,steps,dRows,cols, headers,ids);
        cout << cols << " columns found" << endl;
        argPos=4;
        sortAtomData(ids,pos,types,steps,dRows,ind2Row);
        char *t;
        strcpy(baseName,inpName);
        strcat(baseName,".");
        delete [] ind2Row;
        delete [] ids;
    } else {
        startSave = 0;
        //process as bond/angle data
        if (strstr(inpName,"bData"))
            fwhm[1]=5;
        else if (strstr(inpName,"anData"))
            fwhm[1]=6;
        collectData(inpName,pos,data,steps,dRows,cols,timesteps,headers);
        types = new int[steps*dRows];
        for (int i = 0; i < steps*dRows; ++i) types[i] = 1;
        argPos = 2;
        strcpy(baseName,inpName);
        strcat(baseName,".");
    }
    multipliers = new float[cols];
    units = new char*[cols];
    for (int i = 0; i < cols; ++i) {
        multipliers[i] = atof(argv[argPos++]);
    }
    for (int i = 0; i < cols; ++i) {
        units[i] = new char[strlen(argv[argPos])+1];
        strcpy(units[i],argv[argPos++]);
    }
    res = atoi(argv[argPos++]);
    width = atof(argv[argPos++]);
    cout << "Resolution " << res << " width " << width << endl;
    setDataRange(pos,steps*dRows,dim,min,max,width);
    float *output =
    processData(data,pos,dim,types,fwhm,steps,cols,dRows,multipliers,res,min,max);
    float *total = new float[res*res];
    int l,n=1,stepsSaved;
    char outname[1024];
    ofstream out;
```

```
    stepsSaved = steps - startSave;
    for (int i = 0; i < cols; ++i) {
        cout << "Outputing column " << i << " " << headers[i] << endl;
        for (int j = 0; j < res*res; ++j) total[j] = 0;
        for (int j = startSave; j < steps; ++j) {
            for (int k = 0; k < res*res; ++k) {
                total[k] += output[res*res*steps*i+res*res*j+k];
            }
        }
        cout << "Steps Aggregated for Total" << endl;
        for (int j = 0; j < res*res; ++j) total[j]/=(float)(stepsSaved);
        l = strlen(units[i])+1;
        sprintf(outname,"%s%s.Total.dat",baseName,headers[i]);
        out.open(outname,ios::binary);
        out.write((char*)&n,sizeof(int));
        out.write((char*)&res,sizeof(int));
        out.write((char*)&min,sizeof(float));
        out.write((char*)&max,sizeof(float));
        out.write((char*)&l,sizeof(int));
        out.write((char*)units[i],sizeof(char)*l);
        out.write((char*)timesteps+startSave,sizeof(int));
        out.write((char*)total,sizeof(float)*res*res);
        cout << "Average step file written" << endl;
        out.close();
        cout << "output closed" << endl;
    }
    cout << "Deleting headers and units" << endl;
    for (int i = 0; i < cols; ++i) { delete [] headers[i]; delete [] units[i]; }
    cout << "Deleting total" << endl;
    delete [] total;
    cout << "Deleting units and multipliers, headers pp" << endl;
    delete [] units;
    delete [] multipliers;
    delete [] headers;
    cout << "Deleting positions" << endl;
    delete [] pos;
    cout << "Deleting types, timesteps, data" << endl;
    delete [] types;
    delete [] timesteps;
    delete [] data;
    cout << "Done, exiting" << endl;
    return 0;
}

void collectData(char input[], float* &pos, float* &data, int &steps, int &rows, int &cols, int*
    &timesteps, char** &headers) {
    ifstream in(input);
    char buf[2048];
    in.getline(buf,2048);
    steps = 0;
    rows = 0;
    cols = 0;
    while (strlen(buf)>0) {
        if (strstr(buf,"TIMESTEP")!=false) {
            steps++;
        } else if (strstr(buf,"NUMBER OF")!=false) {
            in >> rows;
            in.getline(buf,2048);
        } else if (strstr(buf,"ITEM: ENTRIES")!=false) {
            if (cols == 0) {
                if (rows == 0) {
                    cout << "Trying to define columns without a row count" << endl;
                    abort();
                }
                char buf2[2048];
```

```
                strcpy(buf2,buf);
                char *tok;
                strtok(buf2," "); strtok(NULL," "); strtok(NULL," "); strtok(NULL," ");
    strtok(NULL," ");
                tok = strtok(NULL," ");
                while (tok) {
                    cols++;
                    tok = strtok(NULL," \n");
                }
                headers = new char*[cols];
                strcpy(buf2,buf);
                strtok(buf2," "); strtok(NULL," "); strtok(NULL," "); strtok(NULL," ");
    strtok(NULL," ");
                tok = strtok(NULL," ");
                int coli = 0;
                while (tok) {
                    headers[coli] = new char[strlen(tok)+1];
                    strcpy(headers[coli++],tok);
                    tok = strtok(NULL," \n");
                }
            }
            for (int i = 0; i < rows; ++i)
                in.getline(buf,2048);
        }
        in.getline(buf,2048);
    }
    pos = new float[3*steps*rows];
    data = new float[cols*steps*rows];
    timesteps = new int[steps];
    in.clear();
    in.seekg(ios::beg);
    in.getline(buf,2048);
    int stepC = 0,posPos,j;
    float *posP = pos, *dataP = data;
    int *tsP = timesteps;
    while (strlen(buf)>0) {
        if (strstr(buf,"TIMESTEP")) {
            in >> *tsP;
            cout << "Reading timestep " << *tsP << endl;
            tsP++;
            in.getline(buf,2048);
        }
        if (strstr(buf,"ITEM: ENTRIES ")!=false) {
            for (int i = 0; i < rows; ++i) {
                for (j = 0; j < 3; ++j) {
                    in >> *posP;
                    posP++;
                }
                for (j = 0; j < cols; ++j) {
                    in >> *dataP;
                    dataP++;
                }
            }
            in.getline(buf,2048);
        }
        in.getline(buf,2048);
    }

}

void setDataRange(float *pos, int rows, int dim[], float &min, float &max, float width) {
    float mean = 0;
    for (int i = 0; i < rows; ++i) {
        mean += pos[dim[0]];
        mean += pos[dim[1]];
```

```
        pos+=3;
    }
    mean/=(float)(2*rows);
    min = mean - width/2.0;
    max = mean + width/2.0;
}

void sortAtomData(int *ids, float* &pos, int* &types, int steps, int dRows, map<int,int>*
    ind2Row) {
    float *newPos = new float[3*dRows*steps];
    int *newTyp = new int[dRows*steps];
    int j,k,oldRow;
    float *pPos; int* pId, *pType;
#pragma omp parallel for num_threads(4) private(posPos, posPos3, oldRow,j,k)
    for (int i = 0; i < steps; ++i) {
        pPos = newPos+3*i*dRows;
        pId = ids+i*dRows;
        pType = newTyp+i*dRows;
        for (j = 0; j < dRows; ++j) {
            if (ind2Row[i].find(*pId)==ind2Row[i].end()) {
                cout << "Couldn't connect id to position row" << endl;
                abort();
            }
            oldRow = ind2Row[i][*pId];
            *pType = types[oldRow];
            for (k = 0; k < 3; ++k)
                pPos[k] = pos[3*oldRow + k];
            pId++; pType++; pPos+=3;
        }
    }
    delete [] pos;
    delete [] types;
    pos = newPos;
    types = newTyp;
}

float* getPos(char input[], int &rows, int &steps, int* &timesteps, int* &types, map<int,int>*
    &ind2row) {
    char buf[1024];
    ifstream in(input);
    in.getline(buf,2048);
    rows = 0; steps = 0;
    while (strlen(buf)>0) {
        if (strstr(buf,"TIMESTEP")!=false)
            steps++;
        else if (rows == 0 && strstr(buf,"NUMBER OF")!=false) {
            in >> rows;
            in.getline(buf,2048);
        } else if (strstr(buf,"ITEM: ATOMS")) {
            for (int i = 0; i < rows; ++i) {
                in.getline(buf,2048);
            }
        }
        in.getline(buf,2048);
    }
    in.clear();
    in.seekg(ios::beg);
    in.getline(buf,2048);
    float *pos = new float[3*steps*rows];
    types = new int[steps*rows];
    timesteps = new int[steps];
    ind2row = new map<int,int>[steps];
    int stepC=0, index;
    float *pPos = pos;
    int posIdx = 0;
```

```
    while (strlen(buf)>0) {
        if (strstr(buf,"TIMESTEP")!=false) {
            in >> timesteps[stepC];
            in.getline(buf,2048);
            cout << "Processing positions step " << stepC << " timestep " << timesteps[stepC] <<
    endl;
        } else if (strstr(buf,"ITEM: ATOMS")!=false) {
            for (int i = 0; i < rows; ++i) {
                in >> index >> types[posIdx] >> pPos[0] >> pPos[1] >> pPos[2];
                ind2row[stepC][index] = posIdx++;
                pPos+=3;
            }
            stepC++;
            in.getline(buf,2048);
        }
        in.getline(buf,2048);
    }
    return pos;
}

void getFWHM(char input[], map<int,float> &fwhm) {
    ifstream in(input);
    char buf[1024];
    in.getline(buf,1024);
    while (strstr(buf,"Pair Coeffs")==false)
        in.getline(buf,1024);
    in.getline(buf,1024);
    in.getline(buf,1024);
    int maxType = 0, tmp;
    while (strlen(buf)>0) {
        tmp = atoi(strtok(buf," \t\n"));
        if (tmp > maxType) maxType = tmp;
        in.getline(buf,1024);
    }
    in.seekg(ios::beg);
    while (strstr(buf,"Pair Coeffs")==false)
        in.getline(buf,1024);
    in.getline(buf,1024);
    in.getline(buf,1024);
    while (strlen(buf)>0) {
        tmp = atoi(strtok(buf," \t\n"));
        strtok(NULL," \t\n");
        fwhm[tmp] = atof(strtok(NULL," \t\n"));
        if (fwhm[tmp]==0) fwhm[tmp] = 2.5;
        in.getline(buf,1024);
    }
    in.close();
    return;
}

float* getAtomData(char input[], int steps, int &rows, int &cols, char** &headers, int* &ids) {
    int stepC = 0;
    ifstream in(input);
    char buf[2048];
    in.getline(buf,2048);
    rows = 0; cols = 0;
    float *data, *pData;
    int i,j,*pId;
    while(strlen(buf)>0) {
        if (rows==0 && strstr(buf,"NUMBER OF")!=false) {
            in >> rows;
            in.getline(buf,2048);
            ids = new int[rows*steps];
            pId = ids;
        }
```

```
        if (strstr(buf,"ITEM: ATOMS")!=false || strstr(buf,"ITEM: ENTRIES")!=false) {
            if (cols==0) {
                if (rows == 0) {
                    cout << "Trying to define columns without a row count" << endl;
                    abort();
                }
                char buf2[2048];
                strcpy(buf2,buf);
                char *tok;
                strtok(buf2," "); strtok(NULL," "); strtok(NULL," ");
                tok = strtok(NULL," ");
                while (tok) {
                    cols++;
                    tok = strtok(NULL," \n");
                }
                headers = new char*[cols];
                strcpy(buf2,buf);
                strtok(buf2," "); strtok(NULL," "); strtok(NULL," ");
                tok = strtok(NULL," ");
                int coli = 0;
                while (tok) {
                    headers[coli] = new char[strlen(tok)+1];
                    strcpy(headers[coli++],tok);
                    tok = strtok(NULL," \n");
                }
                data = new float[cols*rows*steps];
                pData = data;
            }
            for (i = 0; i < rows; ++i) {
                //posPos = rows*stepC+i;
                in >> *pId++;
                for (j = 0; j < cols; ++j) {
                    in >> *pData++;
                }
            }
            in.getline(buf,2048);
            stepC++;
        }
        in.getline(buf,2048);
    }
    return data;
}
```

CUDA Integration Kernel **kernel.cu**

```
#include "cuda_runtime.h"
#include <helper_cuda.h>
#include <map>
#include <iostream>
#include <windows.h>
#include "device_launch_parameters.h"
using namespace std;
#include <stdio.h>
#include <omp.h>

__global__ void compute(float* data, float *pos, int *types, float *pref, float *invc2, int dim1,
    int dim2, int rows, float min, float xr, float *output, int step, int steps, float* mul, int
    col, int cols) {
    int x       = threadIdx.x;
    int res     = blockDim.x;
    float X     = min+xr*(float)x;
    int y       = blockIdx.x;
    float Y     = min+xr*(float)y;
    output  += col*steps*res*res+step*res*res+y*res+x;
```

```
    data    += step*rows*cols;
    pos     += step*rows*3;
    types   += step*rows;
    output[0] = 0;
    float dx,dy,rr;
    int type;
    for (int i = 0; i < rows; ++i) {
        dx = X - pos[dim1];
        dy = Y - pos[dim2];
        rr = dx*dx+dy*dy;
        output[0] += mul[col]*data[col]*pref[*types]*exp(-rr*invc2[*types]);
        data+=cols;
        pos+=3;
        types++;
    }
}

float* processData(float *data, float *pos, int dim[], int *types, map<int,float> &fwhm, int
    steps, int cols, int rows, float multipliers[], int res, float min, float max) {
    int dev;
    cudaGetDevice(&dev);
    float *dOutput,*dData,*dPos,*dPref,*dInvc2,*dLim,*dMuls;
    int *dTypes;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop,dev);
    checkCudaErrors(cudaMalloc((void**)&dOutput,sizeof(float)*steps*cols*res*res));

    checkCudaErrors(cudaMalloc((void**)&dData,sizeof(float)*steps*cols*rows));
    checkCudaErrors(cudaMemcpy(dData,data,sizeof(float)*steps*cols*rows,cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMalloc((void**)&dPos,sizeof(float)*steps*rows*3));
    checkCudaErrors(cudaMemcpy(dPos,pos,sizeof(float)*steps*rows*3,cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMalloc((void**)&dTypes,sizeof(int)*steps*rows));
    checkCudaErrors(cudaMemcpy(dTypes,types,sizeof(int)*steps*rows,cudaMemcpyHostToDevice));

    int fwhmCount = fwhm.rbegin()->first+1;
    float *pref = new float[fwhmCount];
    float *invc2 = new float[fwhmCount];
    float *lim = new float [fwhmCount];
    float c, pi = 4.0*atan(1.0);
    for (map<int,float>::iterator i = fwhm.begin(); i!=fwhm.end(); ++i) {
        c = i->second/(2.0*sqrt(2.0*log(2.0)));
        pref[i->first] = log(256.0)/(2.0*pi*i->second*i->second);
        invc2[i->first] = 1.0/(2.0*c*c);
        lim[i->first] = i->second*sqrt(log(10000.0)/log(16.0));
    }
    checkCudaErrors(cudaMalloc((void**)&dPref,sizeof(float)*fwhmCount));
    checkCudaErrors(cudaMemcpy(dPref,pref,sizeof(float)*fwhmCount,cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMalloc((void**)&dInvc2,sizeof(float)*fwhmCount));
    checkCudaErrors(cudaMemcpy(dInvc2,invc2,sizeof(float)*fwhmCount,cudaMemcpyHostToDevice));

    checkCudaErrors(cudaMalloc((void**)&dMuls,sizeof(float)*cols));
    checkCudaErrors(cudaMemcpy(dMuls,multipliers,sizeof(float)*cols,cudaMemcpyHostToDevice));

    dim3 grid(res);
    dim3 block(res);
    float xr = (max-min)/(float)res;
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    int rowStart, rowEnd, rowStep;
    rowStart = 0; rowEnd = rows; rowStep = rows;
    double startTime, endTime;
    for (int i = 0 ; i < steps; ++i) {
```

```
        cout << "Launching step " << i+1 << endl;
        for (int j = 0; j < cols; ++j) {
                startTime = omp_get_wtime();

        compute<<<grid,block>>>(dData,dPos,dTypes,dPref,dInvc2,dim[0],dim[1],rows,min,xr,dOutput,
    i,steps,dMuls,j,cols);
                checkCudaErrors(cudaThreadSynchronize());
                endTime = omp_get_wtime();
                cout << "Kernel time: " << endTime - startTime << endl;
        }

    }

    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaDeviceSynchronize());
    float *output = new float[cols*steps*res*res];

    checkCudaErrors(cudaMemcpy(output,dOutput,sizeof(float)*res*res*steps*cols,cudaMemcpyDeviceTo
    Host));
    return output;
}
```

'contactModelExe' is responsible for fitting the pressure distribution functions. It computes fits to the Hertz and JKR contact theories, in accordance with Equations 4.5 and 4.6. CUDA libraries are again used to speed the computations, as the two dimensional difference between the fit function and the data set is computed millions of times during the fitting process. This executable relies on two libraries, 'cudaContact' which handles the GPU accelerated computation of the fit function/data set difference, and 'simAnnealHz' which performs the simulated annealing approach to multimodal function minimization, minimizing the difference between the data set and the fit function over the fit function parameter space.

Contact Model Main Executable **contactModelExe.cpp**

```
#include "stdafx.h"
#include <string.h>
#include <string>
#include <iostream>
#include <fstream>
#include <list>
#include <time.h>
#include <iomanip>
#include <float.h>
#include <math.h>
using namespace std;
#include "cudaContact.h"
#include "simAnnealHz.h"
```

```
//p[0] = a, p[1] = p0, p[2] = p0'
float calcJKR(float* data, float* p, int n, float width) {
    float xr = width/(float)n, x, y, pre;
    int center = n/2;
    float ia2 = 1.0/(p[1]*p[2]);
    int i,j;
    float total = 0, *row;
    for (i = 0; i < n; ++i) {
        row = data+n*i;
        x = xr * (float)(center - i);
        for (j = 0; j < n; ++j) {
            y = xr * (float)(center - i);
            pre = (x*x+y*y)*ia2;
            if (pre<1) {
                pre = sqrt(1-pre);
                row[j] = p[1]*pre+p[2]/pre;
                total+=row[j];
            } else {
                row[j] = 0;
            }
        }
    }
    return total;
}

//p[0] = a, p[1] = p0
float calcHz(float* data, float* p, int n, float width) {
    float xr = width/(float)n, x, y, pre;
    int center = n/2;
    float ia2 = 1.0/(p[0]*p[0]);
    float *row;
    int i,j;
    float total = 0;
    for (i = 0; i < n; ++i) {
        row = data+n*i;
        x = xr * (float)(center - i);
        for (j = 0; j < n; ++j) {
            y = xr * (float)(center - j);
            pre = (x*x+y*y)*ia2;
            if (pre<1)
                row[j] = p[1]*(1-pre);
            else
                row[j] = 0;
            total += row[j];
        }
    }
    return total;
}

int _tmain(int argc, _TCHAR* argv[])
{

   int res,l,steps;
   float min,max;
   ifstream in(argv[1],ios::binary);
   if (!in.is_open()) {
       cout << "Couldn't find input file" << endl;
       return 1;
   }
   in.read((char*)&steps,sizeof(int));
   in.read((char*)&res,sizeof(int));
   in.read((char*)&min,sizeof(float));
   in.read((char*)&max,sizeof(float));
   in.read((char*)&l,sizeof(int));
```

```
char *units = new char[l];
in.read(units,sizeof(char)*l);
int *ts1 = new int[steps];
in.read((char*)ts1,sizeof(int)*steps);
float *inp = new float[res*res];
in.read((char*)inp,sizeof(float)*res*res);
float *dev_inp, *dev_fit;
iniSetup(inp,dev_inp,dev_fit,res);
float maxP=FLT_MIN,minP=FLT_MAX;
double mean = 0;
for (int i = 0; i < res*res; ++i) {
    if (inp[i]>maxP) maxP=inp[i];
    if (inp[i]<minP) minP=inp[i];
    mean+=(double)inp[i]*(double)inp[i];
}
if (mean==0) {
    printf("%g %g %g %g %g %g %g\r\n",0,0,0,0,0,0,0);
    return 0;
}
mean/=double(res*res);
double SStot=0;
for (int i = 0; i < res*res; ++i)
    SStot += ((double)inp[i]-mean)*((double)inp[i]-mean);
float limits[8] = {5,(max-min)/2.0,minP,maxP,0,max-min,0,max-min};
double pHz[4] = {(max-min)/4.0,0,(max/min)/2.0,(max/min)/2.0};
float *hzFit = new float[res*res];
int itMax = 1000000;
float tolerance = 1e-7;
list<float> fi;
list<double*> xi;
cout << "Limits on Hz Fit: " << endl;
cout << "a: " << limits[0] << " " << limits[1] << endl;
cout << "p0: " << limits[2] << " " << limits[3] << endl;
fnsimAnneal(dev_inp,dev_fit,res,max-min,pHz,limits,4,itMax,tolerance,fi,xi,SStot,&calcFHz);
double zeros[3]={0,0,0};
float fit=calcFHz(dev_inp,dev_fit,pHz,res,max-min,SStot);
float fit0=calcFHz(dev_inp,dev_fit,zeros,res,max-min,SStot);
cout << "Hz fit: a=" << pHz[0] << " p0=" << pHz[1] << " f=" << fit << " f0=" << fit0 << endl;
cout << "R^2=" << 1.0-fit/SStot << " R^2-0=" << 1.0-fit0/SStot << endl;
cout << itMax << " function calculations, " << fi.size() << " accepted values" << endl;
itMax = 1000000;
cudaMemcpy(hzFit,dev_fit,sizeof(float)*res*res,cudaMemcpyDeviceToHost);
ofstream log("logHz.txt");
log << "a p0 f" << endl;
while (fi.size()>0 && xi.size()>0) {
    log << (*xi.begin())[0] << " " << (*xi.begin())[1] << " " << *fi.begin() << endl;
    xi.pop_front();
    fi.pop_front();
}
ofstream oFit("fit.dat",ios::binary);
oFit.write((char*)&steps,sizeof(int));
oFit.write((char*)&res,sizeof(int));
oFit.write((char*)&min,sizeof(float));
oFit.write((char*)&max,sizeof(float));
oFit.write((char*)&l,sizeof(int));
oFit.write(units,sizeof(char)*l);
oFit.write((char*)ts1,sizeof(int));
oFit.write((char*)hzFit,sizeof(float)*res*res);
oFit.close();
log.close();
float fitHz = fit;
float *jkrFit = new float[res*res];
double pJKR[3] = {(max-min)/4.0,0,0};
limits[2]*=10;
limits[3]*=10;
```

```
   limits[4] = limits[2];
   limits[5] = limits[3];
   fnsimAnneal(dev_inp,dev_fit,res,max-min,pJKR,limits,3,itMax,tolerance,fi,xi,SStot,&calcFJKR);
   fit0 = calcFJKR(dev_inp,dev_fit,zeros,res,max-min,SStot);
   fit = calcFJKR(dev_inp,dev_fit,pJKR,res,max-min,SStot);
   cudaMemcpy(jkrFit,dev_fit,sizeof(float)*res*res,cudaMemcpyDeviceToHost);
   log.open("logJKR.txt");
   log << "a p0 p0` f" << endl;
   while (fi.size()>0 && xi.size()>0) {
       log << (*xi.begin())[0] << " " << (*xi.begin())[1] << " " << (*xi.begin())[2] << " " <<
    *fi.begin() << endl;
       xi.pop_front();
       fi.pop_front();
   }
   log.close();
   ofstream final("final.dat",ios::binary);
   steps = 3;
   int *ts = new int[3];
   ts[0] = 0;
   ts[1] = 1;
   ts[2] = 2;
   cout << "Jkr fit: a=" << pJKR[0] << " p0=" << pJKR[1] << " p0`=" << pJKR[2] << " fit=" << fit
    << " fit0=" << fit0 << endl;
   final.write((char*)&steps,sizeof(int));
   final.write((char*)&res,sizeof(int));
   final.write((char*)&min,sizeof(float));
   final.write((char*)&max,sizeof(float));
   final.write((char*)&l,sizeof(int));
   final.write(units,sizeof(char)*l);
   final.write((char*)&ts,sizeof(int)*steps);
   final.write((char*)hzFit,sizeof(float)*res*res);
   final.write((char*)jkrFit,sizeof(float)*res*res);
   final.write((char*)inp,sizeof(float)*res*res);
   final.close();
   //out.close();
   printf("%g %g %g %g %g %g %g\r\n",pHz[0],pHz[1],pJKR[0],pJKR[1],pJKR[2],1.0-fitHz/SStot,1.0-
    fit/SStot);
   return 0;
}
```

Simulated Annealing Function Minimization Routines — simAnnealHz.cpp

```
// simAnnealHz.cpp : Defines the exported functions for the DLL application.
/*
Modeled after the work of A. Corana, M. Marchesi, C. Martini, and S.Rediella
ACM Transactions on Mathematical Software, Vol 13, No 3, 1987, 262-280
*/

#include "stdafx.h"
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <list>
#include <iostream>
using namespace std;

#include "simAnnealHz.h"

float fRand021() {
    return (float)rand()/(float)(RAND_MAX);
}

float fRandn121() {
    float r = fRand021();
```

```
    return -1.0+2.0*r;
}

// This is an example of an exported function.
SIMANNEALHZ_API void fnsimAnneal(float *input, float *fitData, int res, float width, double *p,
    float *lims, int n, int &itMax, float &tolerance, list<float> &f, list<double*> &x, double
    norm, float (*calcF)(float*,float*,double*,int,float,double))
{
    cout << "Simulated Anneal" << endl;
    //float *data = new float[res*res];
    float rmax = width/2.0;
    //Initialization
    srand(time(NULL));
    int Ns = n*25, Nt = n*25, Ne = 10;
    int i,ii,j,m,k,h;
    i = j = m = k = h = 0;
    float *c = new float[n];
    double *xi;
    double *xopt;
    double *xp;
    float *v = new float[n];
    float *vp = new float[n];
    float *range = new float[n];
    int *nu = new int[n];
    x.push_back(new double[n]);
    xi = *x.rbegin();
    for (ii = 0; ii < n; ++ii) {
        xi[ii] = p[ii];
        c[ii] = 2;
        range[ii] = lims[2*ii+1]-lims[2*ii];
        v[ii] = 0.1*(range[ii]);
        nu[ii] = 0;
    }
    xopt = xi;
    //INITIAL TEMPERATURE DETERMINATION
    cout << "Determining initial temperature" << endl;
    /*xp = new double[n];
    list<float> tests;
    for (ii = 0; ii < n; ++ii) {
        for (int jj = 0; jj < 5; ++jj) {
            for (int kk = 0; kk < n; ++kk) {
                if (kk==ii)
                    xp[kk] = xi[kk]*(0.5+(double)jj/4.0);
                else
                    xp[kk] = xi[kk];
            }
            tests.push_back(calcF(input,fitData,xp,res,width,norm));
        }
    }
    delete [] xp;
    float total = 0;
    for (list<float>::iterator FI=tests.begin(); FI!=tests.end(); ++FI) {
        total+=*FI;
    }
    total/=(float)tests.size();
    float stdev = 0;
    for (list<float>::iterator FI=tests.begin(); FI!=tests.end(); ++FI) {
        stdev += (*FI-total)*(*FI-total);
    }
    stdev/=(float)tests.size();
    stdev = sqrt(stdev);*/
    float T = 0.1*norm;
    cout << "Initial temperature: " << T << endl;
    float beta=1.0/T;
    //tests.clear();
```

```
//END TEMPERATURE DETERMINATION
float rt = 0.85;
float fi,fopt,fp,prb;
list<float> fus;
fi=fopt=calcF(input,fitData,xi,res,width,norm);
f.push_back(fi);
for (ii = 0; ii < Ne; ++ii) {
    fus.push_back(fi);
}
bool end=0;
float frac;
int rejections = 0;
while (i < itMax && k<Ne) {

    cout << "Cycle with new temp started " << T << endl;
    while (m < Nt) { //looping through successive steps using hte same T wiht varying v
        while (j < Ns) { //looping through successive steps using the same v
            h = 0;
            while (h < n) { //looping through the dimensions
                //STEP 1 generate x`
                xp = new double[n];
                for (ii = 0; ii < n; ++ii) {
                    xp[ii] = xi[ii];
                    if (h == ii)
                        xp[ii] += fRandn121()*v[ii];
                }
                //STEP 2 Testing if xp is in the domain, this basically means that
0<a<width/2
                if (xp[h] < lims[2*h] || xp[h] > lims[2*h+1]) {
                    delete [] xp;
                    rejections++;
                    if (rejections>100) {
                        cout << "Seems stuck with v=";
                        for (int ii = 0; ii < n; ++ii) {
                            cout << " " << v[ii];
                        }
                        v[h]*=0.999;
                        cout << endl;
                    }
                    continue;
                }
                rejections = 0;
                //STEP 3 test x`
                //cout << "ib" << endl;
                fp = calcF(input,fitData,xp,res,width,norm);
                if (fp < fi) {
                    f.push_back(fp);
                    x.push_back(xp);
                    fi = fp;
                    xi = xp;
                    ++i; ++nu[h];
                    if (fp < fopt) {
                        xopt = xp;
                        fopt = fp;
                    }
                    //cout << "Kept" << endl;
                } else {
                    prb = exp((fi-fp)*beta);
                    if (fRand021()<prb) {
                        f.push_back(fp);
                        x.push_back(xp);
                        fi = fp;
                        xi = xp;
                        ++i; ++nu[h];
                        //cout << "Kept by prob" << endl;
```

```
                        } else {
                            ++i;
                            //cout << "Tossed" << endl;
                            delete [] xp;
                        }
                    } //END STEP 3
                    //STEP 4
                    ++h;
                }
                ++j; //END STEP 4
            }
            float VN = 0;
            for (ii = 0; ii < n; ++ii) {
                frac = (float)nu[ii]/(float)Ns;
                if (frac>0.6) {
                    vp[ii] = v[ii]*(1.0+(c[ii]/0.4)*(frac-0.6));
                } else if (frac<0.4) {
                    vp[ii] = v[ii]/(1.0+(c[ii]/0.4)*(0.4 - frac));
                } else {
                    vp[ii] = v[ii];
                }
                VN+=vp[ii];
            }
            for (ii = 0; ii < n; ++ii) { v[ii] = min(vp[ii],range[ii]/2.0); nu[ii] = 0; }
            j = 0;
            ++m;
        } //Step 6, temperature change
        T = rt*T;
        beta = 1.0/T;
        fus.push_back(fi);
        fus.pop_front();
        k++;
        m=0;
        //TERMINATION CONDITIONS
        for (list<float>::iterator FI = fus.begin(); FI!=fus.end(); ++FI) {
            if (abs(*FI-fi) > tolerance) {
                end = 0;
                break;
            } else {
                end = 1;
            }
        }
        if (end || (fi-fopt)<tolerance) {  cout << "Termination criteria met" << endl; break; }
        //END TERMINATION CONDITIONS
        if ((Ne-k)==1) { T = T*0.1; beta=1.0/T; };
        ++i;
        xi = xopt;
        fi = fopt;
    }
    for (ii = 0; ii < n; ++ii) {
        p[ii] = xopt[ii];
    }
    itMax = i;
    delete [] v; delete [] vp; delete [] nu; delete [] c;
    return;
}
```

CUDA Fit Computation Function Declarations **cudaContact.h**

```
#ifndef CUCONT
#define CUCONT

#ifndef MIN
#define MIN(x,y) ((x < y) ? x : y)
```

```
#endif

#include "cuda_runtime.h"
#include "cublas_v2.h"
#include <helper_cuda.h>
#include "device_launch_parameters.h"
#include "thrust/reduce.h"
//Upload data to GPU, set pointers
cudaError_t iniSetup(float* inp, float* &dev_inp, float* &dev_fit, int res);
//Determine difference between Hz function and data set
float calcFHz(float* dev_inp, float* dev_fit, double *p, int res, float width, double norm);
//Determine difference between JKR function and data set
float calcFJKR(float* dev_inp, float* dev_fit, double *p, int res, float width, double norm);

void getNumBlocksAndThreads(int n, int maxBlocks, int maxThreads, int &blocks, int &threads);

template <class T>
//Reduction functions calculate integral of difference functions
void reduce(int size, int threads, int blocks, T *d_idata, T *d_odata);
void getNumBlocksAndThreads(int n, int maxBlocks, int maxThreads, int &blocks, int &threads);

template void
reduce<float>(int size, int threads, int blocks,
              float *d_idata, float *d_odata);
#endif
```

CUDA Fit Function **kernel.cu**

```
#include "cudaContact.h"

#include <iostream>
using namespace std;
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

float reduceIt(float *dev_fit, int res);

__global__ void calcFloatingHzFit(float *inp, float *fit, double xr, int cx, int cy, double ia2,
    double p, double c) {
    int res = blockDim.x;
    int x = threadIdx.x;
    int y = blockDim.y*blockIdx.x+threadIdx.y;
    fit += x + res*y;
    inp += x + res*y;
    int xo = cx - x;
    int yo = cy - y;
    double val = *inp;
    double pre = xr*xr*ia2*double(xo*xo+yo*yo);
    if (pre<1.0)
        val -= p*sqrt(1.0-pre);
    *fit = val*val;
}

__global__ void calcJKRFit(float *inp, float *fit, double xr, int center, double ia2, double p,
    double pp, double c) {
   int res = blockDim.x;
   int x = threadIdx.x;
   int y = blockDim.y*blockIdx.x+threadIdx.y;
   fit += x + res*y;
   inp += x + res*y;
   int xo = center - x;
   int yo = center - y;
   double pre = xr*xr*ia2*double(xo*xo+yo*yo);
```

```
   double val = *inp;
   if (pre<0.999)
        val -= p*sqrt(1.0-pre)+pp/sqrt(1.0-pre);
   *fit = val*val;  //minimizing L2 norm, therefore square the difference */
}

__global__ void calcHzFit(float *inp, float *fit, double xr, double center, double ia2, double p,
    double c) {
   int res = blockDim.x;
   int x = threadIdx.x;
   int y = blockDim.y*blockIdx.x+threadIdx.y;
   fit += x + res*y;
   inp += x + res*y;
   double dx = x*xr-center;
   double dy = y*xr-center;
   double val = *inp;
   double pre = ia2*double(dx*dx+dy*dy);
   if (pre<1.0)
    val -= p*sqrt(1.0 - pre);
   *fit = val*val;
}

cudaError_t iniSetup(float* inp, float* &dev_inp, float* &dev_fit, int res) {
    cudaError_t cs;
    int dev;
    cs = cudaGetDevice(&dev);
    cs = cudaMalloc((void**)&dev_inp,res*res*sizeof(float));
    if (cs != cudaSuccess) {
        cout << "Could not allocate input data on GPU" << endl;
        return cs;
    }
    cs = cudaMalloc((void**)&dev_fit,res*res*sizeof(float));
    if (cs != cudaSuccess) {
        cout << "Could not allocate fit data on GPU" << endl;
        return cs;
    }
    cs = cudaMemcpy(dev_inp,inp,sizeof(float)*res*res, cudaMemcpyHostToDevice);
    if (cs != cudaSuccess) {
        cout << "Could not copy input data to GPU" << endl;
        return cs;
    }
    return cs;
}


float calcFloatingHz(float* dev_inp, float* dev_fit, double *p, int res, float width, double
    SStot) {
    dim3 grid(res);
    dim3 block(res);
    while (block.x*block.y*2<1024) {
        block.y*=2;
        grid.x/=2;
    }
    float xr = width/(float)res;
    int xc = int(p[2]/xr);
    int yc = int(p[3]/xr);
    double ia2;
    if (p[0]==0)
        ia2=0;
    else ia2=1.0/(p[0]*p[0]);
    double fwhm = width/4.0;
    double c = fwhm/2.35482;
    calcFloatingHzFit<<<grid,block>>>(dev_inp,dev_fit,xr,xc,yc,ia2,p[1],c);
    return reduceIt(dev_fit,res);
}
```

```
float reduceIt(float *dev_fit, int res) {
    int threads,blocks;
    getNumBlocksAndThreads(res*res,1000000000,512,blocks,threads);
    int s = blocks;
    float *dev_redSh;
    cudaMalloc((void**)&dev_redSh,s*sizeof(float));
    reduce<float>(res*res,threads,blocks,dev_fit,dev_redSh);
    while (s > 1) {
        getNumBlocksAndThreads(s,1000000000,512,blocks,threads);
        reduce<float>(s,threads,blocks,dev_redSh,dev_redSh);
        s = (s + (threads*2-1)) / (threads*2);
    }
    float ret;
    cudaMemcpy(&ret,dev_redSh,sizeof(float),cudaMemcpyDeviceToHost);
    cudaFree(dev_redSh);
    return ret; ///SStot;
}

float calcFHz(float* dev_inp, float* dev_fit, double *p, int res, float width, double SStot) {
    dim3 grid(res);
    dim3 block(res);
    while (block.x*block.y*2<1024) {
        block.y*=2;
        grid.x/=2;
    }
    double xr = (double)width/double(res);
    float center = width/2.0;
    double ia2;
    if (p[0]==0) {
        ia2=0;
    } else {
        ia2=1.0/(p[0]*p[0]);
    }
    double fwhm=width/4.0;
    double c = fwhm/2.35482;
    calcHzFit<<<grid,block>>>(dev_inp,dev_fit,xr,center,ia2,p[1],c);
    return reduceIt(dev_fit,res);
}

float calcFJKR(float* dev_inp, float* dev_fit, double *p, int res, float width, double SStot) {
    dim3 grid(res);
    dim3 block(res);
    while (block.x*block.y*2<1024) {
        block.y*=2;
        grid.x/=2;
    }
    float xr = width/float(res);
    int center = res/2;
    double ia2;
    if (p[0]==0) {
        ia2=0;
    } else {
        ia2=1.0/(p[0]*p[0]);
    }
    double fwhm=width/4.0;
    double c = fwhm/2.35482;
    calcJKRFit<<<grid,block>>>(dev_inp,dev_fit,xr,center,ia2,p[1],p[2],c);
    float ret = 0;
    int threads,blocks;
    getNumBlocksAndThreads(res*res,1000000000,512,blocks,threads);
    int s = blocks;
    float *dev_redSh;
    cudaMalloc((void**)&dev_redSh,s*sizeof(float));
    reduce<float>(res*res,threads,blocks,dev_fit,dev_redSh);
```

```
    while (s > 1) {
        getNumBlocksAndThreads(s,1000000000,512,blocks,threads);
        reduce<float>(s,threads,blocks,dev_redSh,dev_redSh);
        s = (s + (threads*2-1)) / (threads*2);
    }
    cudaMemcpy(&ret,dev_redSh,sizeof(float),cudaMemcpyDeviceToHost);
    cudaFree(dev_redSh);
    return ret; ///SStot;
}
```

CUDA Integral Reduction Routine **reduction.cu**

```
#include "cudaContact.h"

template<class T>
struct SharedMemory
{
    __device__ inline operator       T *()
    {
        extern __shared__ int __smem[];
        return (T *)__smem;
    }

    __device__ inline operator const T *() const
    {
        extern __shared__ int __smem[];
        return (T *)__smem;
    }
};

bool isPow2(unsigned int x)
{
    return ((x&(x-1))==0);
}

unsigned int nextPow2(unsigned int x)
{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    // perform first level of reduction,
    // reading from global memory, writing to shared memory
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;

    // we reduce multiple elements per thread.  The number is determined by the
    // number of active thread blocks (via gridDim).  More blocks will result
    // in a larger gridSize and therefore fewer elements per thread
    while (i < n)
    {
```

```
        mySum += g_idata[i];

        // ensure we don't read out of bounds -- this is optimized away for powerOf2 sized arrays
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];

        i += gridSize;
    }

    // each thread puts its local sum into shared memory
    sdata[tid] = mySum;
    __syncthreads();


    // do reduction in shared mem
    if (blockSize >= 512)
    {
        if (tid < 256)
        {
            sdata[tid] = mySum = mySum + sdata[tid + 256];
        }

        __syncthreads();
    }

    if (blockSize >= 256)
    {
        if (tid < 128)
        {
            sdata[tid] = mySum = mySum + sdata[tid + 128];
        }

        __syncthreads();
    }

    if (blockSize >= 128)
    {
        if (tid <  64)
        {
            sdata[tid] = mySum = mySum + sdata[tid +  64];
        }

        __syncthreads();
    }

    if (tid < 32)
    {
        // now that we are using warp-synchronous programming (below)
        // we need to declare our shared memory volatile so that the compiler
        // doesn't reorder stores to it and induce incorrect behavior.
        volatile T *smem = sdata;

        if (blockSize >=  64)
        {
            smem[tid] = mySum = mySum + smem[tid + 32];
        }

        if (blockSize >=  32)
        {
            smem[tid] = mySum = mySum + smem[tid + 16];
        }

        if (blockSize >=  16)
        {
            smem[tid] = mySum = mySum + smem[tid +  8];
```

```
        }

        if (blockSize >=   8)
        {
            smem[tid] = mySum = mySum + smem[tid +  4];
        }

        if (blockSize >=   4)
        {
            smem[tid] = mySum = mySum + smem[tid +  2];
        }

        if (blockSize >=   2)
        {
            smem[tid] = mySum = mySum + smem[tid +  1];
        }
    }

    // write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}

template <class T>
void reduce(int size, int threads, int blocks, T *d_idata, T *d_odata)
{
    dim3 dimBlock(threads, 1, 1);
    dim3 dimGrid(blocks, 1, 1);

    // when there is only one warp per block, we need to allocate two warps
    // worth of shared memory so that we don't index shared memory out of bounds
    int smemSize = (threads <= 32) ? 2 * threads * sizeof(T) : threads * sizeof(T);

    // choose which of the optimized versions of reduction to launch
    if (isPow2(size))
    {
        switch (threads)
        {
            case 1024:
                reduce6<T, 1024, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 512:
                reduce6<T, 512, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case 256:
                reduce6<T, 256, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case 128:
                reduce6<T, 128, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case 64:
                reduce6<T,  64, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case 32:
                reduce6<T,  32, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case 16:
                reduce6<T,  16, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case  8:
                reduce6<T,   8, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case  4:
                reduce6<T,   4, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
```

```
            case  2:
                reduce6<T,   2, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
            case  1:
                reduce6<T,   1, true><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata, size);
    break;
        }
    }
    else
    {
        switch (threads)
        {
            case 1024:
                reduce6<T, 1024, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 512:
                reduce6<T, 512, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 256:
                reduce6<T, 256, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 128:
                reduce6<T, 128, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 64:
                reduce6<T,  64, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 32:
                reduce6<T,  32, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case 16:
                reduce6<T,  16, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case  8:
                reduce6<T,   8, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case  4:
                reduce6<T,   4, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case  2:
                reduce6<T,   2, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
            case  1:
                reduce6<T,   1, false><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata,
    size); break;
        }
    }
}


void getNumBlocksAndThreads(int n, int maxBlocks, int maxThreads, int &blocks, int &threads)
{

    //get device capability, to avoid block/grid size excceed the upbound
    cudaDeviceProp prop;
    int device;
    checkCudaErrors(cudaGetDevice(&device));
    checkCudaErrors(cudaGetDeviceProperties(&prop, device));

    threads = (n < maxThreads*2) ? nextPow2((n + 1)/ 2) : maxThreads;
    blocks = (n + (threads * 2 - 1)) / (threads * 2);

    if (blocks > prop.maxGridSize[0])
    {
        printf("Grid size <%d> excceeds the device capability <%d>, set block size as %d
```

```
    (original %d)\n",
               blocks, prop.maxGridSize[0], threads*2, threads);

        blocks /= 2;
        threads *= 2;
    }
    blocks = MIN(maxBlocks, blocks);
}
```

APPENDIX D

STRUCTURE CARVING AND NANOGRAFTING CONTROL SOFTWARE

D.1 Structure Carving Script Generation

Indexed star patterns depicted in Figure 2.15-2.17 were generated using a Witec Confocal Microscope, which provides a text based scripting interface with simple commands to move the tip across the surface. Star formation scripts were generated using PHP scripts, with parameters including the size of the star, size of the inner box region, number of patterned lines, as well as the box indices.

Confocal Star Pattern Script Generator **confStarScript.php**

```
<?PHP
    echo "ms(2);";
    echo "sl(10000);";
    $outer = $argv[1];  //first argument is outer dimension of star
    $inner = $argv[2];  //second argument is inner dimension, defines the nanografting area size
    $lines = $argv[3];  //How many lines on given side
    $tSize = $argv[4];  //How big (in microns) the numbers should be
    $n1 = $argv[5];     //First index
    $n2 = $argv[6];     //second index
    if ($lines%2!=0)
        $lines++;
    $oSpace = $outer/$lines;
    $iSpace = $inner/$lines;
    $inside = FALSE;
    $inOff = ($outer-$inner)/2.0;
    $positions = array();
    $positions[] = array(0,0);
    writeNumber($positions,$n1,$tSize);
    writeNumber($positions,$n2,$tSize);
    $positions[] = array(0,0);
    for ($i = 1; $i <= $lines; $i++) {
        if ($inside) {
            $positions[]=array($inOff+$i*$iSpace,$inOff);          //move over
            $positions[]=array($i*$oSpace,0);                //Move out
        } else {
            $positions[]=array($i*$oSpace,0);                //Move over
            $positions[]=array($inOff+$i*$iSpace,$inOff);    //Move in
        }
        $inside = !$inside;
    }
    $positions[] = array($outer-$tSize,0);
    writeNumber($positions,$n1,$tSize);
    writeNumber($positions,$n2,$tSize);
    $inside = FALSE;
    for ($i = 1; $i <= $lines; $i++) {
        if ($inside) {
            $positions[] = array($outer-$inOff,$inOff+$i*$iSpace);  //Move Up
            $positions[] = array($outer,$i*$oSpace);                //Move Out
```

```
        } else {
            $positions[] = array($outer,$i*$oSpace);                //Move Up
            $positions[] = array($outer-$inOff,$inOff+$i*$iSpace);  //Move in
        }
        $inside = !$inside;
    }
    $positions[] = array($outer-$tSize,$outer-$tSize);
    writeNumber($positions,$n1,$tSize);
    writeNumber($positions,$n2,$tSize);
    $positions[] = array($outer,$outer);
    for ($i = 1; $i <= $lines; $i++) {
        if ($inside) {
            $positions[] = array($outer-$inOff-$i*$iSpace,$outer-$inOff);        //Move Right
            $positions[] = array($outer-$i*$oSpace,$outer);
        } else {
            $positions[] = array($outer-$i*$oSpace,$outer);
            $positions[] = array($outer-$inOff-$i*$iSpace,$outer-$inOff);
        }
        $inside = !$inside;
    }
    $positions[] = array(0,$outer-$tSize);
    writeNumber($positions,$n1,$tSize);
    writeNumber($positions,$n2,$tSize);
    $positions[] = array(0,$outer);
    for ($i = 1; $i <= $lines; $i++) {
        if ($inside) {
            $positions[] = array($inOff,$outer-$inOff-$i*$iSpace);
            $positions[] = array(0,$outer-$i*$oSpace);
        } else {
            $positions[] = array(0,$outer-$i*$oSpace);
            $positions[] = array($inOff,$outer-$inOff-$i*$iSpace);
        }
        $inside = !$inside;

    }
    /*foreach ($positions as $p) {
        echo $p[0]." ".$p[1]."\t";
    }*/

    for ($i = 1; $i < count($positions); $i++) {
        //echo $positions[$i-1][0]." ".$positions[$i-1][1]."\t";
        $dx = $positions[$i][0]-$positions[$i-1][0];
        $dy = $positions[$i][1]-$positions[$i-1][1];
        echo "mr ($dx,$dy);\r\n";
    }

    function writeNumber(&$positions,$number,$s) {
        $p = end($positions);
        switch ($number) {
        case 1:
            $positions[] = array($p[0]+0.5*$s,$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 2:
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            $positions[] = array($p[0]+0.5*$s,$p[1]+0.5*$s);
            $positions[] = array($p[0],$p[1]+0.5*$s);
            $positions[] = array($p[0],$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 3:
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            $positions[] = array($p[0]+0.5*$s,$p[1]+0.5*$s);
            $positions[] = array($p[0],$p[1]+0.5*$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]+$s);
```

```
            $positions[] = array($p[0],$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 4:
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            $positions[] = array($p[0]+0.5*$s,$p[1]+0.5*$s);
            $positions[] = array($p[0],$p[1]+0.5*$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 5:
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            $positions[] = array($p[0],$p[1]+0.5*$s);
            $positions[] = array($p[0],$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 6:
            $positions[] = array($p[0],$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]+$s);
            $positions[] = array($p[0],$p[1]+0.5*$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]+0.5*$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        case 7:
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            $positions[] = array($p[0]+0.5*$s,$p[1]+$s);
            $positions[] = array($p[0],$p[1]+$s);
            $positions[] = array($p[0]+0.5*$s,$p[1]);
            break;
        }
    }
?>
```

D.2 Nanografting Control Script

To control the nanografting process as discussed in Section 2.3, a text based interface was developed for the Agilent 5500 AFM that allows for control of tip position, deflection, and speed and that is capable of performing more complex commands. This is centered around the library 'AFMControl', interfaces by an executable 'AFMScripter', detailed below.

AFM Script Control Software **AFMScripter.cpp**

```
// AFMScripter.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <time.h>
#include "picoscript.h"
#include <iostream>
#include <fstream>
#include <strstream>
#include <string>
#include <windows.h>
```

```
using namespace std;
//#include "pressInt.h"
#include "AFMcontrol.h"

int _tmain(int argc, _TCHAR* argv[])
{
    time_t start, end;
    time(&start);
    ifstream input;
    CAFMcontrol ctrl;
    if (argc==1)
        input.open("STD.txt");
    else {
        input.open(argv[1]);
    }
    argc=2;
    for (int i=1; i<argc+1; ++i) {
        while (!input.eof()) {
                string cmd;
                input >> cmd;
                if (cmd.find("#")!=string::npos) {
                        getline(input,cmd);
                        continue;
                }
                int test=ctrl.procComm(cmd,input);
                if (test==0) {
                        cout << "Command " << cmd << " not recognized, script aborting" << endl;
                        break;
                } else if (test==-1)
                        break;
        }
        if (i<argc)
                input.open(argv[i+1]);
    }
    time(&end);
    double x=difftime(end,start);
    cout << "Total time " << x << " seconds" << endl;
    MessageBeep(MB_OK);
    return 0;
}
```

AFM Control Library, Class Declaration **AFMControl.h**

```
#ifdef AFMCONTROL_EXPORTS
#define AFMCONTROL_API __declspec(dllexport)
#else
#define AFMCONTROL_API __declspec(dllimport)
#endif

// This class is exported from the AFMcontrol.dll
class AFMCONTROL_API CAFMcontrol {
public:
    CAFMcontrol(void);

    int procComm(string command, istream& input);
    int procComm2(char* line);
    void mt(char* params);


    //Sets scan size and resolution
    void setScanArea(int res, double size);

    //Moves tip relative to current position at given speed
    bool moveTip(double dx, double dy, double speed);
```

```
//Moves tip to an absolute position in scan area
bool moveTipAbs(double x,double y, double speed);

//Sets tip deflection, if additive, sets relative to current deflection
double setDefl(double deflection, bool additive);

//Initiates a scan
void scan(int res, double speed);

void liftTip(double distance);       //Disengages feedback and lifts it specified distance
void putTipBack();                          //Reengages feedback
void centerServo();                         //Debugging tool
bool drawBox(double size);           //Draws a box, size is in nanometers, speed in um/s

//Draws star pattern with given dimensions and # of lines at specified speed
void drawStar(double outer, double inner, int lines, double speed);

//Handles grid command, feeding it the lines within the grid command
void gridHandle(istream& input);

//Moves the scan area to specified position (absolute)
void moveScanArea(double x, double y);

//Returns the number of parameters expected for a given script command
int paramCount(string command);

//Determines zero force deflection by lifting tip and measuring deflection
double getZForce();

//Draws a filled box with given dimensions, line spacing by rep or CRLS if given
void drawFilledBox(double x, double y, double dy, int rep, double CRLS=0);

//Initiates FD spectroscopy centered at current tip position spanning given range
double perfFD(double range, double maxF = 0);

//Sets center of scan area, repeat of moveScanArea
void setOffset(double x, double y);

//performs scan with increasing load, width defines number of lines scanned at a given load
void scanLoadControl(double loadIni, double loadFin, int width);

//Connects and disconnects to the pico library
void discon();
void connect();

//return 1 if feedback is on and tip is not approaching (i.e. things are ready)
bool engaged();

//Returns the current applied load
double getLoad();
//Return current tip z position
double getZPosition();

//Agilent library class used by this guy
PicoScript* pico;

//parameters
double radius;      //Stores tip radius, in nm
double k;           //Stores tip spring constant, in N/m
double FD;          //Stores force distance relationship, in V/um
double eps;         //Stores epsilon parameter used in contact radius calcs
double zeroF;       //Stores deflection of tip when not in contact with surface
bool useF;          //Flag indicating whether deflection inputs will be read as nN or V
bool useZF;         //Flag indicating whether or not the non-contact tip deflection will be
```

```
    used as a reference point

};

extern AFMCONTROL_API int nAFMcontrol;

AFMCONTROL_API int fnAFMcontrol(void);
```

AFMControl Command Parsing Functions — commandParse.cpp

```
#include "stdafx.h"
#include <math.h>
#include <iostream>
#include <fstream>
#include <string>
#include <picoscript.h>
#include <sstream>
using namespace std;
//#include "pressInt.h"
#include "AFMcontrol.h"

int CAFMcontrol::procComm(string command, istream& input) {
    cout << "Processing command " << command << endl;
    if (command.find("scanarea")!=string::npos) {
        double width; int res;
        input >> width >> res;
        cout << "Setting scan width " << width << "um and Resolution " << res << " points/line"
    << endl;
        setScanArea(res,width);
    } else if (command.find("setdef")!=string::npos) {
        bool additive=0; double def;
        if (command.find("R")!=string::npos)
                additive=1;
        input >> def;
        setDefl(def,additive);
    } else if (command.find("slp")!=string::npos) {
        double sl;
        input >> sl;
        cout << "Sleeping " << sl/1000.0 << " seconds" << endl;
        if (integ) {
                int res=pico->GetScanParameter(scanXPixels);
                double x=pico->GetStatus(statusScanPixel)*pico->GetScanParameter(scanSize)/res;
                double y=pico->GetStatus(statusScanLine)*pico->GetScanParameter(scanSize)/res;
                double fz=getLoad();
                //sl-=PI->holdInt(x,y,fz,sl);
        }
        Sleep(sl);
    } else if (command.find("rotate")!=string::npos) {
        double rot;
        input >> rot;
        cout << "Setting scan angle " << rot << " degrees" << endl;
        pico->SetScanParameter(scanAngle,rot);
    } else if (command.find("appr")!=string::npos) {
        cout << "Sample approaching" << endl;
        pico->Motor(motorApproach);
        Sleep(5000);
        pico->WaitFor(statusApproachState,0);
        cout << "Sample approached" << endl;
    } else if (command.find("withd")!=string::npos) {
        double dist;
        input >> dist;
        if (dist!=0)
                pico->SetMotorParameter(motorWithdrawDistance,dist);
        pico->Motor(motorWithdraw);
```

```
        cout << "Sample withdrawn";
    } else if (command.find("setsp")!=string::npos) {
        double speed;
        input >> speed;
        cout << "Speed set to " << speed << " um/s" << endl;
        pico->SetScanParameter(scanTipSpeed,speed/1000000.0);
    } else if (command.find("mt")!=string::npos) {
        double dx, dy, speed;
        input >> dx >> dy;
        if (command.find("S")!=string::npos) {
                input >> speed;
        } else {
                speed=pico->GetScanParameter(scanTipSpeed)*1000000;
        }
        if (command.find("R")!=string::npos)
                moveTip(dx,dy,speed);
        else
                moveTipAbs(dx,dy,speed);
    } else if (command.find("sbias")!=string::npos) {
        double cb=0;
        if (command.find("r")!=string::npos)
                cb=pico->GetServoParameter(servoBias);
        double b;
        input >> b;
        cout << "Setting servo bias to " << cb+b << endl;
        pico->SetServoParameter(servoBias,cb+b);
    } else if (command.find("lt")!=string::npos) {
        double d;
        input >> d;
        cout << "Feedback loop off, lifting tip " << d << " um off the surface" << endl;
        liftTip(d);
    } else if (command.find("dbFill")!=string::npos) {
        double x, y, dy=0, CRLS=0;
        int rep=0;
        input >> x >> y;
        if (command.find("CRLS")!=string::npos)
                input >> CRLS;
        else
                input >> dy;
        if (command.find("r")!=string::npos) {
                cout << "reps" << endl;
                input >> rep;
                cout << rep << endl;
        }
        cout << "x " << x << " y " << y << " dy " << dy << endl;
        drawFilledBox(x,y,dy,rep,CRLS);
    } else if (command.find("db")!=string::npos) {
        double s;
        input >> s;
        cout << "Drawing box with dimension " << s << " nm." << endl;
        if (!drawBox(s))
                return -1;
    } else if (command.find("fbOn")!=string::npos) {
        putTipBack();
    } else if (command.find("star")!=string::npos) {
        double outer, inner, speed;
        int lines;
        input >> outer >> inner >> lines >> speed;
        cout << "Drawing star with outer dimension " << outer << " um, inner dimensino " << inner
    << " um, at speed " << speed << " (um/s) with " << lines << " lines per side" << endl;
        drawStar(outer, inner, lines, speed);
    } else if (command.find("grid")!=string::npos) {
        gridHandle(input);
    } else if (command.find("scanPos")!=string::npos) {
        double x,y;
```

```
        input >> x >> y;
        cout << "Setting stage position to " << x << ' ' << y << " um" << endl;
        moveScanArea(x,y);
    } else if (command.find("res")!=string::npos) {
        int res;
        input >> res;
        if (res<=13)
                res=pow(2.0,res);
        pico->SetScanParameter(scanXPixels,res);
    } else if (command.find("setTip")!=string::npos) {
        input >> k >> FD;
        cout << "Spring constant set to " << k << " N/m and Force-Distance Relationship to " <<
    FD << " V/um" << endl;
    } else if (command.find("setContact")!=string::npos) {
        input >> radius >> eps;
        cout << "Spring radius set to " << radius << " nm, epsilon set to " << eps << endl;
    } else if (command.find("useForce")!=string::npos) {
        if (k!=0 && FD!=0) {
                useF=(!useF);
                cout << "Use force for deflection inputs set to ";
                if (useF)
                        cout << "on";
                else
                        cout << "off";
                cout << endl;
        } else {
                cout << "Spring constant and FD relation must be set, see command setTip" <<
    endl;
        }
    } else if (command.find("getZForce")!=string::npos) {
        getZForce();
    } else if (command.find("useZForce")!=string::npos) {
        useZF=(!useZF);
    } else if (command.find("ss")!=string::npos) {
        if (command.find("up")!=string::npos) {
                pico->Scan(scanStartUp);
        } else if (command.find("down")!=string::npos) {
                pico->Scan(scanStartDown);
        } else {
                return 0;
        }
        if (command.find("W")!=string::npos) {
                pico->WaitFor(statusScanning,0);
        }
    } else if (command.find("zRange")!=string::npos) {
        double range;
        input >> range;
        range/=(2.0*1000000.0);
        pico->SetServoParameter(servoTopographyRange,range);
        pico->Servo(servoOptimize);
        Sleep(100);
        pico->Servo(servoOptimize);
        Sleep(100);
        pico->Servo(servoOptimize);
        Sleep(100);
        pico->Servo(servoOptimize);
        Sleep(100);
        pico->Servo(servoOptimize);
        Sleep(100);
    } else if (command.find("fd")!=string::npos) {
        double range;
        input >> range;
        perfFD(range);
    } else if (command.find("offset")!=string::npos) {
        double x,y;
```

```
        input >> x >> y;
        setOffset(x,y);
    } else if (command.find("loadScan")!=string::npos) {
        double lIni,lFin;
        int width;
        input >> lIni >> lFin >> width;
        scanLoadControl(lIni,lFin,width);
    } else if (command.find("zPos")!=string::npos) {
        cout << "Z position: " << getZPosition() << endl;
    } else {
        return 0;
    }
    return 1;
}

int CAFMcontrol::paramCount(string command) {
    if (command.find("scanarea")!=string::npos) {
        return 2;
    } else if (command.find("setdef")!=string::npos) {
        return 1;
    } else if (command.find("slp")!=string::npos) {
        return 1;
    } else if (command.find("withd")!=string::npos) {
        return 1;
    } else if (command.find("setsp")!=string::npos) {
        return 1;
    } else if (command.find("mt")!=string::npos) {
        if (command.find("S")!=string::npos)
                return 3;
        else
                return 2;
    } else if (command.find("sbias")!=string::npos) {
        return 1;
    } else if (command.find("lt")!=string::npos) {
        return 1;
    } else if (command.find("star")!=string::npos) {
        return 4;
    } else if (command.find("scanPos")!=string::npos) {
        return 2;
    } else if (command.find("res")!=string::npos) {
        return 1;
    } else if (command.find("dbFill")!=string::npos) {
        if (command.find("r")!=string::npos)
                return 4;
        else
                return 3;
    } else if (command.find("db")!=string::npos) {
        return 1;
    } else if (command.find("setTip")!=string::npos) {
        return 2;
    } else if (command.find("setContact")!=string::npos) {
        return 2;
    } else {
        return 0;
    }
}
```

AFM Control Function Definitions **AFMcontrol.cpp**

```
// AFMcontrol.cpp : Defines the exported functions for the DLL application.
//

#include "stdafx.h"
#include <math.h>
```

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <string>
#include <picoscript.h>
#include <sstream>
using namespace std;
#include "AFMcontrol.h"


// This is an example of an exported variable
AFMCONTROL_API int nAFMcontrol=0;

// This is the constructor of a class that has been exported.
// see AFMcontrol.h for the class definition
CAFMcontrol::CAFMcontrol()
{
    pico=new PicoScript;
    useF=0;
    FD=0;
    eps=0;
    radius=0;
    k=0;
    zeroF=0;
    useZF=0;
    integ=0;
    //PI=NULL;
    return;
}

void CAFMcontrol::connect() {
    if (pico==NULL)
        pico=new PicoScript;
}

void CAFMcontrol::discon() {
    if (pico!=NULL) {
        delete pico;
        pico=NULL;
    }
}

void CAFMcontrol::setScanArea(int res, double size)
{
    pico->SetScanParameter(scanSize,size/1000000.0);
    pico->WaitFor(statusStageMoveInProgress,0);
    if (res<=13)
        res=int(pow(2.0,res));
    pico->SetScanParameter(scanXPixels,res);
    Sleep(2000);
}

bool CAFMcontrol::moveTip(double dx, double dy, double speed) // dx dy in nm, speed in m/s, moves
    tip in direction specified by straight line
{

    pico->SetScanParameter(scanTipSpeed,speed/1000000.0);
    double ss=pico->GetScanParameter(scanSize)*1000000.0;
    int res=pico->GetScanParameter(scanXPixels);
    double spatRes=double(res)/ss;
    //Get relative movement distances in pixels
    double db[2];
    int d[2];
    db[0]=(dx/1000.0)*spatRes;
    db[1]=(dy/1000.0)*spatRes;
```

```
    for (int i=0; i<2; ++i) {
        if (db[i]-floor(db[i])>0.5)
                d[i]=ceil(db[i]);
        else
                d[i]=floor(db[i]);
    }
    //Get Actual distances traveled
    dx=(ss/double(res))*double(d[0]);
    dy=(ss/double(res))*double(d[1]);
    double totD=sqrt(pow(dx,2)+pow(dy,2));
    //Get Current Location
    int cX=pico->GetStatus(statusScanPixel);
    int cY=pico->GetStatus(statusScanLine);
    if (cX+d[0]-1>res || cY+d[1]-1>res || cX+d[0]<0 || cY+d[1]<0) {
        cout << "Attempting to move outside of scan range" << endl;
        abort();
    }
    double slLoss=0;
    cout << "Moving tip " << dx << '(' << d[0] << ") " << dy << '(' << d[1] << "), distance " <<
    totD << endl;
    //cout << " um, speed:" << speed << " um/s, sleep:" << int(ceil(1000*totD/speed+50.0)) << "
    ms" << endl;
    pico->SetTipPosition(cX+d[0],cY+d[1]);
    if (integ) {
        double fz=getLoad();
        double ix[2],iy[2];
        ix[0]=1000.0*cX/spatRes;
        ix[1]=1000.0*(cX+d[0])/spatRes;
        iy[0]=1000.0*cY/spatRes;
        iy[1]=1000.0*(cY+d[1])/spatRes;
        //slLoss=PI->moveInt(ix,iy,speed,fz);
    }
    //cout << "Sleeping " << int(ceil(1000*totD/speed))+10 << " milliseconds" << endl;
    Sleep(int(ceil(1000*totD/speed)-slLoss)+10);
    //cout << "done" << endl;
    return 1;
}

bool CAFMcontrol::moveTipAbs(double x,double y,double speed) //x, y in nm, speed in um/s
{
    //Get Scan Parameters, set tip speed for motion
    pico->SetScanParameter(scanTipSpeed,speed/1000000.0); //set scan speed for tip move
    double ss=pico->GetScanParameter(scanSize)*1000000.0; //get scan size in um
    if ((x-1.0)/1000.0>ss || x<0) {
        cout << "X coordinate greater than scan area, aborting" << endl;
        abort();
    } else if ((y-1.0)/1000.0>ss || y<0) {
        cout << "Y coordinate greater than scan area, aborting" << endl;
        abort();
    }
    //cout << "Scan size " << ss << endl;
    int res=pico->GetScanParameter(scanXPixels);
    //Desired location in pixels
    double dbX=(x/(ss*1000.0))*double(res);
    double dbY=(y/(ss*1000.0))*double(res);
    int nx, ny;
    if (dbX-floor(dbX)>0.5)
        nx=ceil(dbX);
    else
        nx=floor(dbX);
    if (dbY-floor(dbY)>0.5)
        ny=ceil(dbY);
    else
        ny=floor(dbY);
    //Get Current Location
```

```
    int cX=pico->GetStatus(statusScanPixel);
    int cY=pico->GetStatus(statusScanLine);
    //Get distance to be traveled
    int iX=nx-cX;
    int iY=ny-cY;
    //distances traveled in um
    double dx=(ss/double(res))*double(iX);
    double dy=(ss/double(res))*double(iY);
    double totD=sqrt(pow(dx,2)+pow(dy,2));
    double slLoss=0;
    double spatRes=double(res)/ss;
    pico->SetTipPosition(nx,ny);
    if (integ) {
        double fz=getLoad();
        double ix[2],iy[2];
        ix[0]=1000.0*cX/spatRes;
        ix[1]=1000.0*nx/spatRes;
        iy[0]=1000.0*cY/spatRes;
        iy[1]=1000.0*ny/spatRes;
        //slLoss=PI->moveInt(ix,iy,speed,fz);
    }
    cout << "Moving tip " << dx << '(' << iX << ") " << dy << '(' << iY << "), distance " << totD
    << endl;
    Sleep(int(ceil(1000*totD/speed)-slLoss)+10);
    //pico->WaitFor(statusStageMoveInProgress,0);
    return 1;
}

void CAFMcontrol::moveScanArea(double x, double y) {
    pico->SetScanParameter(scanXOffset,x/1000000.0);
    pico->SetScanParameter(scanYOffset,y/1000000.0);
    pico->WaitFor(statusStageMoveInProgress,0);
    return;
}

double CAFMcontrol::setDefl(double deflection, bool additive) //sets new deflection and returns
    old, additive applies new deflection in addition to current defl
{
    double tmp=pico->GetServoParameter(servoSetpoint);
    if (useF) {
        if (!useZF) {
                cout << "Cannot set absolute deflection by force without using zero contact
    force, see getZForce" << endl;
                abort();
        }
        if (additive) {
                double currF=(tmp-zeroF)*1000*k/FD;
                deflection+=currF;
        }
        deflection=FD*(deflection/(1000*k));
        deflection+=zeroF;
    } else if (additive) {
        deflection+=tmp;
    } else if (useZF) {
        deflection=zeroF+deflection;
    }
    //cout << "Setting deflection to " << tmp*additive+deflection << endl;
    pico->SetServoParameter(servoSetpoint,deflection);
    Sleep(200);
    return deflection;
}

void CAFMcontrol::liftTip(double distance)
{
    double currPos=getZPosition();
```

```
    //cout << "Current Z position: " << currPos << endl;
    pico->SetServoParameter(servoActive,0);
    pico->SetServoParameter(servoZDirect,(currPos+distance)/1000000.0);
    Sleep(100);
    return;
}

void CAFMcontrol::putTipBack()
{
    pico->SetServoParameter(servoActive,1);
    Sleep(100);
}

void CAFMcontrol::centerServo()
{
    pico->SetServoParameter(servoActive,0);
    pico->SetServoParameter(servoZDirect,0.0);
    pico->SetServoParameter(servoActive,1);
}

bool CAFMcontrol::drawBox(double size)
{
    cout << "Drawing Box ";
    double speed=pico->GetScanParameter(scanTipSpeed)*1000000.0;
    if (!moveTip(size,0,speed))
        return 0;
    if (!moveTip(0,size,speed))
        return 0;
    if (!moveTip(-size,0,speed))
        return 0;
    if (!moveTip(0,-size,speed))
        return 0;
    return 1;
}

void CAFMcontrol::drawStar(double outer, double inner, int lines, double speed)
{
    bool wasInt=integ;
    if (integ)
        integ=0;
    liftTip(2.5);
    Sleep(1000);
    setScanArea(8192,outer);
    moveTipAbs(0,0,2.0);
    putTipBack();
    Sleep(500);
    //Need deflection setting line here
    Sleep(500);
    bool inward=1;
    for (int i=0; i<lines; i++)
    {
        if (inward) {
                moveTipAbs(1000.0*(outer-
    inner)/2.0+1000.0*double(i)*(inner/double(lines)),1000.0*(outer-inner)/2.0, speed);
                moveTipAbs(1000.0*(outer-inner)/2.0+1000.0*double(i+1)*(inner/double(lines)),
    1000.0*(outer-inner)/2.0, speed);
        } else {
                //cout << "Drawing out" << endl;
                moveTipAbs(1000.0*double(i)*outer/double(lines), 0, speed);
                moveTipAbs(1000.0*double(i+1)*outer/double(lines), 0, speed);
        }
        inward=!inward;
    }
    for (int i=0; i<lines; i++)
    {
```

```
        if (inward) {
                moveTipAbs(1000.0*(0.5*(outer-inner)+inner),1000.0*(outer-
    inner)/2.0+1000.0*double(i)*(inner/double(lines)),speed);
                moveTipAbs(1000.0*(0.5*(outer-inner)+inner),1000.0*(outer-
    inner)/2.0+1000.0*double(i+1)*(inner/double(lines)),speed);
        } else {
                moveTipAbs(1000.0*outer,1000*double(i)*(outer/double(lines)),speed);
                moveTipAbs(1000.0*outer,1000*double(i+1)*(outer/double(lines)),speed);
        }
        inward=!inward;
    }
    for (int i=lines; i>0; i--)
    {
        if (inward) {
                moveTipAbs(1000.0*(outer-
    inner)/2.0+1000*double(i)*(inner/double(lines)),1000.0*(0.5*(outer-inner)+inner), speed);
                moveTipAbs(1000.0*(outer-inner)/2.0+1000*double(i-
    1)*(inner/double(lines)),1000.0*(0.5*(outer-inner)+inner), speed);
        } else {
                moveTipAbs(1000.0*double(i)*(outer/double(lines)),1000.0*outer,speed);
                moveTipAbs(1000.0*double(i-1)*outer/double(lines),1000.0*outer,speed);
        }
        inward=!inward;
    }
    for (int i=lines; i>0; i--)
    {
        if (inward) {
                moveTipAbs(1000.0*(outer-inner)/2.0,1000.0*(outer-
    inner)/2.0+1000.0*double(i)*(inner/double(lines)),speed);
                moveTipAbs(1000.0*(outer-inner)/2.0,1000.0*(outer-inner)/2.0+1000.0*double(i-
    1)*(inner/double(lines)),speed);
        } else {
                moveTipAbs(0,1000.0*double(i)*(outer/double(lines)),speed);
                moveTipAbs(0,1000.0*double(i-1)*(outer/double(lines)),speed);
        }
        inward=!inward;
    }
    integ=wasInt;
}

#define index(col,row,depth,columns,rows) (col+columns*row+columns*rows*depth)

void CAFMcontrol::gridHandle(istream& input) {
    int lines, count, p=0;
    input >> lines >> count;
    string* commands = new string[lines];
    stringstream* params=new stringstream[lines*count*count];
    cout << lines*count*count << " total streams" << endl;
    int vary[2];vary[0]=0;vary[1]=0;
    double varies[2][2];
    double width=0;
    input >> lines >> count;
    input.clear(ios::goodbit);
    cout << "Creating " << count << " grid with " << lines << " commands." << endl;
    input >> commands[0];
    //First check for variable parameters
    for (int p=0; p<3; p++) {
        cout << "FC=" << commands[0] << endl;
        if (commands[0].compare("vb")==0) {
                vary[p]=1;
                input >> varies[p][0] >> varies[p][1] >> commands[0];
        } else if (commands[0].compare("vset")==0) {
                vary[p]=2;
                input >> varies[p][0] >> varies[p][1] >> commands[0];
        } else if (commands[0].compare("vsp")==0) {
```

```
                vary[p]=3;
                input >> varies[p][0] >> varies[p][1] >> commands[0];
        } else if (commands[0].find("width")!=string::npos) {
                input >> width >> commands[0];
                //cout << "Width of elements set to " << width << " microns" << endl;
        } else
                break;
    }
    //Prepare variable values
    cout << "Prepping variable vars" << endl;
    double* vp1;
    double* vp2;
    if (vary[0]!=0) {
        vp1 = new double[count];
        for (int i=0; i<count; i++) {
                vp1[i]=varies[0][0]+double(i)/double(count-1)*(varies[0][1]-varies[0][0]);
        }
    }
    if (vary[1]!=0) {
        vp2 = new double[count];
        for (int i=0; i<count; i++) {
                vp2[i]=varies[1][0]+double(i)/double(count-1)*(varies[1][1]-varies[1][0]);
        }
    }
    //Get Commands
    cout << "Getting commands" << endl;
    bool setdef=0, sb=0, setsp=0;
    for (int i=1; i<=lines; i++) {
        //cout << "Command " << i-1 << ' ' << commands[i-1] << endl;
        if (commands[i-1].find("sbias")!=string::npos && !sb) {
                sb=1;
                double param;
                input >> param;
                if (vary[0]==1) {
                        //cout << "Sbias branch " << varies[0][0] << " " << varies [0][1] <<
    endl;
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++)
                                        params[index(i-1,j,k,lines,count)] << vp1[j];
                        }
                } else if (vary[1]==1) {
                        //cout << "Sbias branch " << varies[0][0] << " " << varies [0][1] <<
    endl;
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++)
                                        params[index(i-1,k,j,lines,count)] << vp2[j];
                        }
                } else {
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++)
                                        params[index(i-1,j,k,lines,count)] << param;
                        }
                }
        } else if (commands[i-1].find("setsp")<5 && !setsp) {
                setsp=1;
                //cout << "Setsp branch " << vary[0] << ' ' << vary[1] << endl;
                double param;
                input >> param;
                if (vary[0]==3) {
                        //cout << "Setsp branch " << varies[0][0] << " " << varies [0][1] <<
    endl;
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++) {
                                        params[index(i-1,j,k,lines,count)] << vp1[j];
                                }
```

```
                        }
                } else if (vary[1]==3) {
                        //cout << "Setsp branch " << varies[1][0] << " " << varies [1][1] <<
endl;
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++) {
                                        params[index(i-1,k,j,lines,count)] << vp2[j];
                                }
                        }
                } else {
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++) {
                                        cout << "Setting command(" << index(i-1,j,k,lines,count)
<< ") " << commands[i-1] << " with value " << param << endl;
                                        params[index(i-1,j,k,lines,count)] << param;
                                }
                        }
                }
        } else if (commands[i-1].find("setdef")<5 && !setdef) {
                setdef=1;
                cout << "SetDef Branch" << endl;
                double param;
                input >> param;
                if (vary[0]==2) {
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++)
                                        params[index(i-1,j,k,lines,count)] << vp1[j];
                        }
                } else if (vary[1]==2) {
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++)
                                        params[index(i-1,k,j,lines,count)] << vp2[j];
                        }
                } else {
                        for (int j=0; j<count; j++) {
                                for (int k=0; k<count; k++) {
                                        //cout << "Setting command(" << index(i-
1,j,k,lines,count) << ") " << commands[i-1] << " with value " << param << endl;
                                        params[index(i-1,j,k,lines,count)] << param;
                                }
                        }
                }
        } else {
                //cout << "All others branch" << endl;
                int pcount=paramCount(commands[i-1]);
                double* ps;
                ps = new double[pcount];
                for (int j=0; j<pcount; j++)
                        input >> ps[j];
                for (int j=0; j<count; j++) {
                        for (int k=0; k<count; k++) {
                                for (int p=0; p<pcount; p++) {
                                        //cout << "Setting command(" << index(i-
1,j,k,lines,count) << ") " << commands[i-1] << " with value " << ps[p] << endl;
                                        params[index(i-1,j,k,lines,count)] << ps[p] << ' ';
                                }
                        }
                }
                delete [] ps;
        }
        if (i<lines)
                input >> commands[i];
}
cout << "Commands Gotten" << endl;
//Prepare positions
```

```
    double sarea=pico->GetScanParameter(scanSize)*1000000.0;
    double* x;
    x = new double[count];
    double spacing=(sarea-width*count)/double(count);
    if (spacing < 0) {
        cout << "Too many objects, too wide, not enough room" << endl;
        abort();
    }
    cout << "Gridpoints" << endl;
    for (int i=0; i<count; i++) {
        x[i]=(spacing/2.0+(width+spacing)*double(i))*1000.0;
        cout << x[i] << ' ';
    }
    cout << endl;
    cout << "Commands are ";
    for (int i=0; i<lines; i++) {
        cout << commands[i] << ' ';
    }
    for (int i=0; i<count; i++) {
        for (int j=0; j<count; j++) {
                int J;
                if (i%2==0)
                        J=j;
                else
                        J=count-j-1;
                cout << "Grid point " << x[i] << ' ' << x[J] << endl;
                double currSp=pico->GetScanParameter(scanTipSpeed)*1000000.0;
                moveTipAbs(x[J],x[i],currSp);
                double currSetP=pico->GetServoParameter(servoSetpoint);
                for (int k=0; k<lines; k++) {
                        params[index(k,i,J,lines,count)].seekg(0,ios::beg);
                        if (procComm(commands[k],params[index(k,i,J,lines,count)])!=1)
                                return;
                }
                pico->SetServoParameter(servoSetpoint, currSetP);
                //pico->SetServoParameter(servoBias, currBias);
                pico->SetScanParameter(scanTipSpeed,currSp/1000000.0);
        }
    }
    cout << "Exiting grid handler" << endl;
    if (vary[0]!=0) {
        delete [] vp1;
    }
    if (vary[1]!=0) {
        delete [] vp2;
    }
    delete [] x;
    delete [] commands;
    delete [] params;
}

void CAFMcontrol::drawFilledBox(double x, double y, double dy, int rep, double CRLS) {
    double fz;
    if (CRLS!=0) {
        if (radius==0 || eps==0 || k==0 || eps==0 || useF==0) {
                cout << "Tip radius, spring constant, and FD relation must be set, as well as
    contact epsilon and zero contact force, to use CRLS" << endl;
                abort();
        }
        fz=getLoad();
        double contactR=pow(3.0*radius*fz/(4.0*eps),(1.0/3.0));
        dy=contactR/CRLS;
    }
    bool forward=1;
    int count=0;
```

```
    double currSp=pico->GetScanParameter(scanTipSpeed)*1000000.0;
    while (count<=rep) {
        cout << "Drawing box " << count+1 << endl;
        double totaly=0;
        while (totaly<y) {
                if (forward)
                        moveTip(x,0,currSp);
                else
                        moveTip(-x,0,currSp);
                totaly+=dy;
                forward=(!forward);
                if (totaly>y)
                        break;
                else {
                        if (count%2==0)
                                moveTip(0,dy,currSp);
                        else
                                moveTip(0,-dy,currSp);
                }
        }
        count++;
    }
    if (CRLS!=0)
        cout << "Linestep was " << dy << " nm, force=" << fz << endl;
}

double CAFMcontrol::getZForce() {
    double currSp=pico->GetServoParameter(servoSetpoint);
    pico->SetServoParameter(servoSetpoint,-10.0);
    Sleep(500);
    zeroF = 0;
    for (int i = 0; i < 5; ++i) {
        zeroF+=pico->GetStatus(statusRawDefl);
        Sleep(201);
    }
    zeroF /= 5.0;
    pico->SetServoParameter(servoSetpoint,currSp);
    cout << "Zero Force Deflection obtained: " << zeroF << " V" << endl;
    return zeroF;
}

double CAFMcontrol::perfFD(double range, double maxF) {
    double currPos=getZPosition();
    cout << currPos;
    double FDTop, FDBot;
    FDTop=(currPos+(range/2.0))/1000000.0;
    FDBot=(currPos-(range/2.0))/1000000.0;
    pico->SetSpectroscopyParameter(spectroscopyStart,FDTop);
    pico->SetSpectroscopyParameter(spectroscopyEnd,FDBot);
    if (maxF > 0 && useZF) {
        pico->SetSpectroscopyParameter(spectroscopyMaxLimitEnable,true);
        if (useF) {
                double TS = k*1000.0/FD;
                maxF *= TS;
                pico->SetSpectroscopyParameter(spectroscopyMaxLimit,maxF + zeroF);
        } else {
                pico->SetSpectroscopyParameter(spectroscopyMaxLimit,maxF + zeroF);
        }
    } else {
        pico->SetSpectroscopyParameter(spectroscopyMaxLimitEnable,false);
    }
    pico->Spectroscopy(spectroscopySweepStart);
    while (pico->GetStatus(statusSpectroscopySweeping))
        Sleep(100);
    return currPos;
```

```
}

double CAFMcontrol::getZPosition() {
    double currPos = 0;
    for (int i = 0; i < 5; ++i) {
        currPos += pico->GetStatus(statusZPosition);
        Sleep(201);
    }
    currPos /= 5.0;
    return currPos*1000000.0;
}

void CAFMcontrol::setOffset(double x, double y) {
    pico->SetScanParameter(scanXOffset,x/1000000.0);
    pico->SetScanParameter(scanYOffset,y/1000000.0);
    pico->WaitFor(statusStageMoveInProgress,0);
}

void CAFMcontrol::scanLoadControl(double loadIni, double loadFin, int width) {
    getZForce();
    useZF=1;
    int res=pico->GetScanParameter(scanYPixels);
    if (res%width!=0) {
        cout << "Width must be divisor of resolution, fix it" << endl;
        abort();
    }
    int steps=res/width;
    cout << steps << " steps" << endl;
    double* loads=new double[steps];
    for (int i=0; i<steps; ++i) {
        loads[i]=loadIni+i*(loadFin-loadIni)/(steps-1);
        cout << loads[i] << endl;
    }
    setDefl(loads[0],0);
    pico->Scan(scanStartUp);
    while (pico->GetStatus(statusScanning)) {
        int line=pico->GetStatus(statusScanLine);
        double relPos=double(line)/double(res);
        double step=relPos*double(steps);
        int STEP=floor(step);
        setDefl(loads[STEP],0);
        Sleep(20);
    }
    delete [] loads;
}

double CAFMcontrol::getLoad() {
    if (!useF) {
        cout << "Trying to determine load when it is not available!" << endl;
        abort();
    }
    if (!pico->GetScanParameter(scanTipLift))
        return 0.0;
    double curDef=pico->GetServoParameter(servoSetpoint);
    return (curDef-zeroF)*k*1000/FD;
}

bool CAFMcontrol::engaged() {
    return pico->GetServoParameter(servoActive) && pico->GetStatus(statusApproachState)!=1;
}
```